Distribution
Category UC-411

SAND98-8206 (revised)
Unlimited Release
First Printed November 1997

Jess, The Java Expert System Shell

http://herzberg.ca.sandia.gov/jess
Ernest J. Friedman-Hill
Distributed Computing Systems
Sandia National Laboratories
Livermore, CA

Version 4.3 (December 3rd, 1998)

ABSTRACT

This report describes Jess, an expert system shell written entirely in Java. Jess supports the development of rule-based expert systems which can be tightly coupled to code written in the powerful, portable Java language. The syntax of the Jess language is discussed, and a comprehensive list of supported functions is presented. A guide to calling Java functions from Jess, and to extending Jess by writing Java code, is also included.

1 Introduction

Jess is an expert system shell written entirely in Java. Jess was originally a clone of the essential core of CLIPS, but has begun to acquire a Java-influenced flavor of its own. With Jess, you can conveniently give your Java applets and applications the ability to "reason." In describing Jess, I am going to describe much of CLIPS itself, but the reader may want to have a copy of the CLIPS manuals at hand. See the CLIPS site for more information.

Jess 4.3 is compatible with all versions of Java starting with version 1.0.2. It is compatible with version 1.1, although while compiling you will see warnings about deprecated methods. Such is the price of compatibility! *Note:* Jess 4.3 is the last version that will be compatible with Java 1.0.2; future versions of Jess will not work with anything less than Java 1.1. That means that the "deprecated" warnings will disappear.

Jess is a work in progress; more features are constantly being added. The order will be determined in part by what folks seem to want most, what I need Jess to do, and how much time I have to spend on it. See Version History for a list of what's new in this version of Jess and see What's New in This Release for a quick overview.

There is a Jess email discussion list you can join. To get information about the jess-users list, send a message to majordomo@sandia.gov containing the text

```
  help
  info jess-users
  end
```

as the **body** of the message.

This is the final release of Jess 4.3. Although this version has undergone extensive testing, It's always possible that there are bugs. Please report any that you find to me at ejfried@ca.sandia.gov so I can fix them for a later release.

1.1 Getting Started With Jess

1.1.1 Unpacking the Distribution

If you download Jess for UNIX, you can extract the files using tar and uncompress:

```
uncompress Jess-4.3.tar.Z
tar xf Jess-4.3.tar
```

If you downloaded Jess for Windows, you get a .zip file which should be unzipped using a Win32-aware unzip program like WinZip. Don't use PKUNZIP since it cannot handle long file names.

When Jess is unpacked, you should have a directory named `Jess43/`. Inside this directory should be the following files:

`README.html`	This file
`jess/`	A directory containing the `jess` package. There are many source files in here that implement Jess's inference engine. Others implement a number of Jess GUIs and command-line interfaces. `Main.java` implements the Jess command-line interface. `Console.java` is a very simple GUI console for Jess; `ConsoleApplet.java` is an applet version of the same.
`jess/view`	Java source implementing the optional Jess `view` command.
`jess/reflect`	Java source implementing the optional Jess commands that let you create and manipulate Java objects from Jess.
`examples/`	A directory of tiny example Jess files.
`jess/examples`	A directory of more complicated examples, containing example Java source files.
`index.html`	A web page containing the Jess example applet. It may need to be edited.
`Makefile`	A simple makefile for Jess.

1.1.2 Compiling Jess

Jess comes as a set of Java source files. You'll need to compile them first. If you have a `make` utility (any UNIX `make`; or `nmake` or GNU `make` on Win32), you can just run `make` and the enclosed makefile will build everything. You might have to edit it a bit first. Otherwise the commands:

```
  javac jess/*.java (UNIX)
```

or

```
  javac jess\*.java (Win32)
```

would work just fine, given that you have a Java compiler like Sun's JDK, and that `Jess43/` is your current directory. If you have problems, be sure that the directory in which the jess subdirectory appears is on your `CLASSPATH`; this may mean including `.` (dot). *Don't* try to compile from inside the `Jess43/jess/` directory; it won't work. You can use either a Java 1.0.2 or a Java 1.1 compiler to compile Jess. The resulting code runs on either 1.0 or 1.1 VMs. Note that if you use a 1.1 compiler, you will see some warning about deprecated methods. It is safe to ignore these warnings. Jess also seems to work with Java 1.2.

There are a number of optional source files in the subdirectories `Jess43/jess/view`/, `Jess43/jess/reflect`/ and `Jess43/jess/examples`/ that aren't compiled if you follow the instructions above. These files define the optional debugging command `view`, the reflection commands `new`, `call`, `set`, `get`, `set-member`, and `get-member`, and the Java object matching commands `defclass` and `definstance`. They can be compiled only with Java 1.1 or later. If you have such a compiler, then you can issue a command like:

```
  javac jess/*.java jess/view/*.java jess/reflect/*.java jess/examples/*/*.java
(Unix)
```

or

```
  javac jess\*.java jess\view\*.java jess\reflect\*.java jess\examples\pumps\*.java
jess\examples\simple\*.java (Win32)
```

to compile everything (or use the Makefile, of course).

Again, *don't* set your current directory to, for example, Jess43/jess/examples/pumps/ to compile the pumps example: it will *not* work. The compiler will report all sorts of errors about classes not being found and the jess package not being found. Compile everything from the Jess43 directory. I can't stress this enough: this is by far the most common problem people have in getting started with Jess!

1.1.3 Jess Example Programs

There are several example programs for you to try, including `fullmab.clp`, `zebra.clp`, and `wordgame.clp`. `fullmab.clp` is a version of the classic Monkey and Bananas problem. To run it yourself from the command line, just type:

```
  java jess.Main examples/fullmab.clp (Unix)
```

or

```
  java jess.Main examples\fullmab.clp (Win32)
```

and the problem should run, producing a few screens of output. Any file of Jess code can be run this way. Many simple CLIPS programs will also run unchanged in Jess. Note that giving Jess a file name on the command line is like using the `batch` command in CLIPS. Therefore, you need to make sure that the file ends with:

```
  (reset)
  (run)
```

or nothing will happen. The `zebra.clp` and `wordgame.clp` programs are two classic CLIPS examples selected to show how Jess deals with tough situations. These examples both generate huge numbers of partial pattern matches, so they are slow and use up a lot of memory. They each can take some time to run, depending on your computer. Other examples include `sticks.clp` (an interactive game) and `frame.clp` (a demo of building a graphical interface using Jess's Java integration capabilities).

In the jess/examples/* subdirectories, you will find some more complex examples, all of which contain both Java and Jess code. As such, these are generally examples of how to tie Jess and Java together. The *Pumps* examples is a full working program that demonstrates how Jess rules can react to the properties of Java Beans.

1.1.4 Command-line Interface

Jess has an interactive command-line interface. Just type `java jess.Main` to get a `Jess>` prompt. To execute a file of CLIPS code from the command prompt, use the `batch` command:

```
  Jess> (batch myfile.clp)
    (lots of output)
```

You can use the Jess `system` command to invoke an editor from the Jess command line to edit a file of Jess code before reading it in with `batch`. `system` also helps to allow non-Java programmers to integrate Jess with other applications. Given that you have an editor named `notepad` on your system, try:

```
  Jess> (system notepad README &)
    TRUE
```

The & character makes the editor run in the background. Omitting it will keep the system command from returning until the called program exits.

The class `jess.Console` is a graphical verison of the Jess command-line interface. Output appears in a scrolling window. Type `java jess.Console` to try it.

1.1.5 Jess as an Applet

The class `jess.ConsoleApplet` is a generic Jess applet that uses the same display as the `jess.Console` class. It can be used in general question-and-answer situations simply by embedding the applet class on a Web page. The applet accepts two applet parameters. The value of an `INPUT` parameter will be interpreted as a Jess program to run when starting up. Note that when this program halts, the Jess prompt will appear in the applet window. The applet also accept a `COMPACT` parameter. If present, `ConsoleApplet` will contain only a bare-bones version of Jess (no optional functions will be loaded).

1.2 What's New in This Release

If you've used Jess before, this section will help you get started quickly with this version. Jess 4.3 offers quite a few new features and user-visible changes:

- Since the beginning, the standard Jess console and Applet classes have been combined into one class. In the 4.0 release, the curiously named QuizDisplay class continued this ill-advised economy, combining an Applet, an application, -and- a ReteDisplay subclass in one! This has been remedied: These classes have been broken out into jess.Main, jess.Console, jess.ConsoleApplet, and jess.ConsoleDisplay. The old 'Monkey and Banana' applet with the flashing color lights has been (alas) removed and is no longer supported.
- The `Userfunction` interface has changed slightly. `name()` returns a String. RU.getAtom() and `RU.putAtom()` are gone; there is no longer any need for them. A few public methods here and there that used to accept integers or return integers now return Strings or accept Strings as arguments; for example, the `name()` methods of all the `Def(X)` classes now return String.
- The `GlobalContext` class has disappeared, and the way execution contexts are managed internally has been simplified. This should be an invisible change for 99.9% of users.
- Adding input routers has been slightly complicated, as you can specify how the `read` command should behave with a given router.
- A manual section has been added that better explains writing main() for a Java app thet embeds Jess.
- You can now supply a function call as a slot default value in a deftemplate; and you can supply defaults for multislots. Furthermore, the new default-dynamic deftemplate slot attribute lets you specify that a default function call value for a slot be evaluated each time a deftemplate fact is asserted.
- ***Important!*** The way that defglobals behave in the face of the (reset) command has changed to match CLIPS' behavior. The set-reset-globals and get-reset-globals commands have been added to let you modify this. See their documentation for a better description of the changes.
- The set-strategy command has been added, bringing user-selectable conflict resolution strategies to Jess.
- The try command has been added, bringing exception handling to Jess!
- The new agenda command lets you see what will happen next.
- The set-salience-evaluation command lets you use dynamic rule salience (a fairly advanced feature.)
- The `unique` conditional element in new. ***Important!*** You can no longer use the atom 'unique' as the head of a deftemplate as a result. `Unique` lets you give hints to the Rete engine and can theoretically result in speedups of up to 50%; in practice I've observered real speedups of 20-30%; for examples, see the `wordgame` and `zebra` sample programs.
- The `store` and `fetch` functions (both in Jess and as Java member functions of the `jess.Rete` class) let you easily pass values between Jess and Java, from simple types to Java objects.
- The `jess.reflect.Canvas` class has been added. It lets you write GUIs in Jess that include drawing and painting, without writing any Java code.

---

2 The Jess Language

Jess is an interpreter for a rule language borrowed from CLIPS. Given CLIPS's heritage (strongly influenced by systems written in LISP), this rule language is basically a small, idiosyncratic version of LISP, making Jess a LISP interpreter written in Java. I will briefly describe this language here; more information can be gotten from the CLIPS manuals themselves.

I'm using an extremely informal notation to describe syntax. Basically strings in <angle-brackets> are some kind of data that must be supplied; things in [square brackets] are optional, things ending with + can appear one or more times, and things ending with * can appear zero or more times.

In general, input to Jess is free-format. Newlines are generally not significant and are treated as whitespace.

In the example dialogs, you type what appears after the *Jess>* prompt. The system responds with the text in **bold.**

2.1 Atoms

The atom or symbol is a core concept of the Jess language. Atoms are very much like identifiers in other languages. A Jess atom can contain letters, numbers, and the following punctuation: `$*=+/<>_?#.` . An atom may not begin with a number; it may begin with some punctuation marks (some have special meanings as operators when they appear at the start of an atom). The best atoms consist of letters, numbers, underscores, and dashes; dashes are traditional word separators. The following are all valid atoms:

```
foo first-value contestant#1 _abc
```

2.2 Numbers

Jess parses numbers using the Java `StreamTokenizer` class. Therefore, it accepts only simple floating point and integer numbers. It does not accept scientific or engineering notation. The following are all valid numbers:

```
3 4. 5.643
```

2.3 Strings

Character strings in Jess are denoted using double quotes (`""`). Backslashes (`\`) can be used to escape embedded quote symbols. The following are all valid strings:

```
"foo" "Hello, World" "\"Nonsense,\" he said firmly."
```

2.4 Lists

The fundamental unit of syntax in Jess is the list. A list always consists of an enclosing set of parentheses and zero or more atoms, numbers, strings, or other lists. The following are valid lists:

```
(+ 3 2) (a b c) ("Hello, World") () (deftemplate foo (slot bar))
```

The first element of a list (the *car* of the list in LISP parlance) is often called the list's *head* in Jess.

2.5 Comments

Programmer's comments in Jess begin with a semicolon (`;`) and extend to the end of the line of text. Here is an example of a comment:

```
; This is a list
(a b c)
```

2.6 Functions

Jess contains a large number of built-in functions that you may call. More functions are provided as extensions. You can write your own functions in the Jess language (see Deffunctions) or in Java (see Extending Jess with Java).

Function calls in Jess use a prefix notation. A list whose head is an atom that is the name of an existing function can be evaluated as an expression. For example, an expression that uses the `+` function to add the numbers `2` and `3` would be written `(+ 2 3)`. When evaluated, the value of this expression is the number `5` (not a list containing the single element `5`!). In general, expressions are recognized as such and evaluated in context when appropriate. You can type expressions at the `Jess>` prompt. Jess evaluates the expression and prints the result:

```
Jess> (+ 2 3)
  5
Jess> (+ (+ 2 3) (* 3 3))
  14
```

Note that arithmetic results may be returned as floating-point numbers or as integers, depending on the types of the arguments.

Jess implements only a small subset of CLIPS functions as intrinsic functions that are built into Jess and cannot be removed. All of these have been designed to function as much like their CLIPS counterparts as possible. On the other hand, I'm supplying implementations for many more CLIPS functions, and lots of functionality specific to Jess, as 'Userfunctions' - external functions written in Java that you can plug into Jess. All of the included Userfunctions are installed into the command-line version of Jess by default; you can pick and choose in your own applications. In applets, in particular, you may want to include only the Userfunctions you need, to keep the size of the applet down. (see Extending Jess with Java for information about doing this.)

Here is the complete list of functions shipped with Jess 4.3, both intrinsic and optional:

```
* ** + - / < <= <> = > >= abs agenda and assert assert-string bag batch
bind build call clear close complement$ create$ defclass definstance
delete$ div e engine eq eq* eval evenp exit exp explode$ external-addressp
facts fetch first$ float floatp foreach format gensym* get get-member
get-reset-globals get-salience-evaluation get-var halt if implode$ insert$
integer integerp intersection$ jess-version-number jess-version-string
length$ lexemep list-function$ load-facts load-function load-package log
log10 lowcase max member$ min mod modify multifieldp neq new not nth$
numberp oddp open or pi ppdefrule printout random read readline replace$
reset rest$ retract retract-string return round rules run save-facts set
set-member set-reset-globals set-salience-evaluation set-strategy setgen
socket sqrt store str-cat str-compare str-index str-length stringp
sub-string subseq$ subsetp sym-cat symbolp system time try undefinstance
undefrule union$ unwatch upcase view watch while
```

All these functions are described in detail in the Jess Function Guide. Note that the distinction between intrinsic functions and Userfunctions is mostly an academic one; intrinsic functions are all written as Java classes that implement the same `Userfunction` interface that user-supplied classes do. The only real difference is whether Jess will start up without them; the intrinsics are required because they're loaded in by code in the jess.Funcall class. To find out if a function is intrinsic, see its entry in the Function Guide below.

2.7 Variables

Programming variables in Jess are atoms that begin with the question mark (`?`) character. The question mark is part of the variable's name. A normal variable can refer to a single atom, number, or string. A variable whose first character is instead a `$` (for example, `$?X`) is a *multivariable*, which can refer to a special kind of list called *multifield*. You assign to any variable using the `bind` function:

```
(bind ?x "The value")
```

Multifields are generally created using special multifield functions like `create$` and can then be bound to multivariables:

```
(bind $?grocery-list (create$ eggs bread milk))
```

Variables need not (and cannot) be declared before their first use (except for Defglobals).

2.8 Constructs

Besides expressions and multifields, the Jess language includes another kind of special list called a *construct*. A construct is a list that defines something to the Jess system itself. For example, the `deffunction` construct is used to define functions (see Deffunctions). A construct evaluates to `TRUE` if it was accepted by Jess or `FALSE` if it was not.

2.9 Deffunctions

The `deffunction` construct is used to define functions that you can then call from Jess. A `deffunction` construct looks like this:

```
(deffunction <function-name> [<doc-comment>] (<parameter>*)
<expr>*
[<return-specifier>])
```

The `<function-name>` must be an atom. Each `<parameter>` must be a variable name (all functions use pass-by-value semantics). The optional `<doc-comment>` is a double-quoted string that can describe the purpose of the function. There may be an arbitrary number of `<expr>` expressions. The optional `<return-specifier>` gives the return value of the function. It can either be an explicit use of the `return` function or it can be any value or expression. Control flow in `deffunctions` is achieved via the special control-flow expressions `foreach`, `if`, and `while`. The following is a `deffunction` that returns the numerically larger of its two numeric arguments:

```
(deffunction max (?a ?b)
  (if (> ?a ?b) then
      (return ?a)
   else
      (return ?b)))
```

Note that this could have also been written as:

```
(deffunction max (?a ?b)
  (if (> ?a ?b) then
      ?a
   else
      ?b))
```

2.10 Facts

Jess maintains a list of facts or information about the current state of the system. Facts may be *ordered* or *unordered*. Ordered facts are merely lists whose head must be an atom:

```
(temperature 98.6)
(shopping-list bread milk paper-towels)
(start-processing)
```

Unordered facts are structured. They contain a definite set of *slots* which must be accessed by name. While ordered facts can be used without prior definition, unordered facts must be defined using the `deftemplate` construct (see Deftemplates).

Facts are placed on the fact list by the `assert` function. You can see the current fact list using the `facts` function. You can remove (`retract`) a fact from the fact list if you know its fact ID. For example:

```
Jess> (assert (foo bar))
  <Fact-0>

Jess> (facts)
  f-0   (foo bar)
  For a total of 1 facts.
  TRUE
Jess> (retract 0)
  TRUE
Jess> (facts)
  For a total of 0 facts.
  TRUE
```

2.11 Deftemplates

To define an unordered fact, use the `deftemplate` construct:

```
(deftemplate <deftemplate-name> [<doc-comment>]
  [(slot <slot-name> [(default <value>)]
                        [(default-dynamic <value>)]
                        [(type <typespec>)])]+)
```

The `<deftemplate-name>` is the head of the facts that will be created using this `deftemplate`. There may be an arbitrary number of slots. The `<slot-name>` must be an atom. The `default` slot qualifier states that the default value of a slot in a new fact is given by `<value>`; the default is the atom `nil`. The 'default-dynamic' version will evaluate the given function each time a new fact using this template is asserted. The 'type' slot qualifier is accepted (for compatibility with CLIPS) but is ignored by Jess.

As an example, defining the following `deftemplate`:

```
  (deftemplate automobile
    "A specific car."
    (slot make)
    (slot model)
    (slot year)
    (slot color (default white)))
```

would allow you to define facts like this:

```
  Jess> (assert (automobile (make Chrysler) (model LeBaron) (year 1997)))
    <Fact-0>
  Jess> (facts)
    f-0   (automobile (make Chrysler) (model LeBaron) (year 1997) (color white))
    For a total of 1 facts.
    TRUE
```

Note that the car is white by default. Also note that any number of additional automobiles could also be simultaneously asserted onto the fact list using this `deftemplate`.

A given slot in a `deftemplate` fact can normally hold only one value. If you want a slot that can hold multiple values, use the `multislot` keyword instead:

```
  (deftemplate box
    (slot location)
    (multislot contents))

  (assert (box (location kitchen) (contents spatula sponge frying-pan)))
```

2.12 Defclasses

A `defclass` construct basically lets you use a Java Bean as a deftemplate. Almost any Java object can be made into a Bean. This is a very powerful feature of Jess that lets it reason about the state of objects connected to the physical world. `defclass` will be documented later, after we've explained some of the prerequisites.

2.13 Deffacts

The deffacts construct is a handy way to define a list of facts that should be made true when the Jess system is started or reset.

```
  (deffacts <deffacts-name>
    [<doc-comment>]
    <fact>+)
```

The primary purpose of the `<deffacts-name>` is documentation. A `deffacts` instance can contain any number of facts. Any unordered facts in a `deffacts` instance must have previously been defined via a `deftemplate` construct when the `deffacts` is parsed. The following is a valid deffacts construct:

```
  (deffacts automobiles
    (automobile (make Chrysler) (model LeBaron) (year 1997))
    (automobile (make Ford) (model Contour) (year 1996))
    (automobile (make Nash) (model Rambler) (year 1948)))
```

2.14 Definstances

What `deffacts` are to `deftemplates`, `definstances` are to `defclasses.` While a `deffacts` construct defines an initial set of facts to the Rete engine, the `definstance` construct tells Jess that one particular Java object should be treated as if it were a fact and be matched by `deftemplate` patterns defined in `defclass` constructs. Again, we'll defer discussion until we're in a better position to understand the mechanics involved.

2.15 Defrules

The main purpose of an expert shell like Jess is to support the execution of rules. Rules in Jess are somewhat like the IF...THEN... statements of other programming languages. In operation, Jess constantly tests to see if any of the IFs become true, and executes the corresponding THENs. (Actually, it doesn't work quite this way, but this is a good way to imagine things. See How Jess Works for an explanation closer to the truth.) The intelligence embedded in an intelligent rule-based system is encoded in the rules. The `defrule` construct is used to define a rule to Jess:

```
  (defrule <defrule-name>
    [<doc-comment>]
    [<salience-declaration>]
    [[<pattern-binding> <- ] <pattern>]*
    =>
    <action>*)
```

Basically, a rule consists of a list of patterns (the IF part on the rule's left-hand-side or LHS) and a list of actions (the THEN part on the rule's right-hand-side or RHS). The patterns are matched against the fact list. When facts are found that match all the patterns of a rule, the rule becomes activated, meaning it may be fired (have its actions executed).

> **Note:** The patterns on rule LHSs are matched against the fact-list as if they were facts - they are *NOT* function calls! The following rule does *NOT* work:
>
> ```
>    (defrule wrong-rule
>       (eq (+ 2 2) 4)
>       =>
>       (printout t "Just as I thought, 2 + 2 = 4!" crlf))
> ```
>
> This rule will NOT fire just because the function call (eq (+ 2 2) 4) would evaluate to true. Instead, Jess will try to find a fact on the fact-list that looks like (eq 4 4). Unless you have previously asserted such a fact, this rule will *NOT* be activated and will not fire. If you want to fire a rule based on the evaluation of a function, you can use the test CE.

An activated rule may become deactivated before firing if the facts that matched its patterns are retracted, or removed from the fact list, while it is waiting to be fired. Here is an example of a simple rule:

```
  (defrule example-1
    "Announce 'a b c' facts"
    (a b c)
     =>
    (printout t "Saw 'a b c'!" crlf))
```

To see this rule in action, enter it at the `Jess>` prompt, assert the fact `(a b c)`, then the `run` command to start the Jess engine. You'll get some interesting additional information by first issuing the `watch all` command:

```
  Jess> (clear)
    TRUE
  Jess> (watch all)
    TRUE
  Jess> (defrule example-1
          "Announce 'a b c' facts"
          (a b c)
```

```
         =>
          (printout t "Saw 'a b c'!" crlf))
  example-1: +1+1+1+1+t
  TRUE
Jess> (assert (a b c))
   ==> Activation: example-1 : f-0
   ==> (a b c)
   <Fact-0>
Jess>  (run)
  FIRE example-1 f-0
  Saw 'a b c'!
  TRUE
Jess>
```

When you enter the rule, you see the sequence of symbols `+1+1+1+1+t`. This tells you something about the way that Jess compiled the rule you wrote into the internal rule representation. Then when you assert the fact, Jess responds by telling you that the new fact was assigned the numeric fact identifier 0 (`f-0`), and that it is an ordered fact with head `a` and additional fields `b` and `c`. Then it tells you that the rule example-1 is activated by the fact f-0, that fact you just entered. When you type the `run` command, you see an indication that your rule has been fired, including a list of the relevant fact IDs. The line `"Saw 'a b c'!"` is the result the execution of your rule.

Multiple activated rules are fired in order of salience (see Salience). Within a given salience value, the order in which rules will fire is given by the current *conflict resolution strategy*. See the `set-strategy` command for details. You can see the list of activated, but not yet fired, rules with the command.

If all the patterns of a rule had to be given literally as above, Jess would not be very powerful. However, patterns can also include wildcards and various kinds of *predicates* (comparisons and boolean functions). You can specify a variable name instead of a value for a field in any of a rule's patterns (but not the pattern's head). A variable matches any value in that position within a rule. For example, the rule:

```
(defrule example-2
  (a ?x ?y)
  =>
  (printout t "Saw 'a " ?x " " ?y "'" crlf))
```

will be activated each time any fact with head `a` having two fields is asserted: `(a b c)`, `(a 1 2)`, `(a a a)`, and so forth. As in the example, the variables thus matched in the patterns (or LHS) of a rule are available in the actions (RHS) of the same rule.

Each such variable field in a pattern can also include any number of tests to qualify what it will match. Tests follow the variable name and are separated from it and from each other by ampersands. (The variable name itself is actually optional.) Tests can be:

- A literal value (in which case the variable matches *only* that value).
- Another variable (which must have been matched earlier in the rule's LHS). This will constrain the field to contain the same value as the variable was first bound to.
- A colon (`:`) followed by a function call, in which case the test succeeds if the function returns the special value `TRUE`. These are called *predicate constraints*.
- An equals sign (`=`) followed by a function call. In this case the field must match the return value of the function call. These are called *return value constraints*. Note that both predicate constraints and return-value constraints can refer to variables bound elsewhere in this or any preceding pattern in the same defrule. ***Note:*** pretty-printing a rule containing a return value contstraint will show that it has been transformed into an equivalent predicate constraint.
- Any of the other options preceded by a tilde (`~`), in which case the sense of the test is reversed (inequality or false).

Here's an example of a rule that uses several kinds of tests:

```
(defrule example-3
  (not-b-and-c ?n1&~b ?n2&~c)
```

```
    (different ?d1 ?d2&~?d1)
    (same ?s ?s)
    (more-than-one-hundred ?m&:(> ?m 100))
    =>
    (printout t "Found what I wanted!" crlf))
```

The first pattern will match a fact with head `not-b-and-c` with exactly two fields such that the first is not `b` and the second is not `c`. The second pattern will match any fact with head `different` and two fields such that the two fields have different values. The third pattern will match a fact with head `same` and two fields with identical values. The last pattern matches a fact with head `more-than-one-hundred` and a single field with a numeric value greater than 100.

A few more details about patterns: you can match a field without binding it to a variable by omitting the variable name and using just a question mark (`?`) as a placeholder. You can match any number of fields using a multivariable (one starting with `$?`):

```
  Jess> (defrule example-4
    (grocery-list $?list)
    =>
    (printout t "I need to buy " $?list crlf))
    TRUE
  Jess> (assert (grocery-list eggs milk bacon))
    TRUE
  Jess> (run)
    I need to buy (eggs milk bacon)
    TRUE
```

And finally, to access a global variable on the left-hand side of a rule, you must use the get-var function.

2.15.1 Pattern bindings.

Sometimes you need a handle to an actual fact that helped to activate a rule. For example, when the rule fires, you may need to retract or modify the fact. To do this, you use a pattern-binding variable:

```
  (defrule example-5
    ?fact <- (command "retract me")
    =>
    (retract ?fact))
```

The variable (`?fact`, in this case) is assigned the fact ID of the particular fact that activated the rule.

2.15.2 Salience.

Rules normally fire in an order related to which rules were most recently activated. See the `set-strategy` command for details. To force certain rules to always fire first or last, rules can include a salience declaration:

```
  (defrule example-6
    (declare (salience -100))
    (command exit-when-idle)
    =>
    (printout t "exiting..." crlf))
```

Declaring a low salience value for a rule makes it fire after all other rules of higher salience. A high value makes a rule fire before all rules of lower salience. The default salience value is zero. Salience values can be integers, global variables, or function calls. See the set-salience-evaluation command for details about when such function calls will be evaluated.

2.15.3 `Not` patterns.

A pattern can be enclosed in a list with `not` as the head. In this case, the pattern is considered to match if a fact which matches the pattern is not found. For example:

```
  (defrule example-7
     (person ?x)
     (not (married ?x))
     =>
     (printout t ?x " is not married!" crlf))
```

Note that a `not` pattern cannot contain any variables that are not bound before that pattern (since a `not` pattern does not match any facts, it cannot be used to define the values of any variables!) You can use *blank* variables, however (a blank variable is a bare `?` or `$?`). A `not` pattern can similarly not have a pattern binding.

2.15.4 The `test` conditional element (CE).

A pattern with `test` as the head is special; the body consists not of slot tests but of a single function which is evaluated and whose truth determines whether the pattern matches. For example:

```
  (defrule example-8
     (person (age ?x))
     (test (> ?x 30))
     =>
     (printout t ?x " is over 30!" crlf))
```

Note that a `test` pattern, like a `not`, cannot contain any variables that are not bound before that pattern. `test` and `not` may be combined:

```
          (not (test (eq ?X 3)))
```

is equivalent to:

```
          (test (neq ?X 3))
```

2.15.5 The `unique` conditional element.

A pattern can be enclosed in a list with `unique` as the head. This is a hint to Jess that only one fact could possibly satisfy a given pattern, given matches for the preceding patterns in that rule. Here's an example:

```
(defrule unique-demo
   (tax-form (social-security-number ?num))
   (unique (person (social-security-number ?num) (name ?name)))
   =>
   (printout t "Auditing " ?name "..." crlf))
```

Here the `unique` CE is providing a hint to Jess that only one person can have a given Social Security number. Given this knowledge, Jess knows that once it has found the person that matches a given tax form, it doesn't need to look any further. In practice, this can result in performance gains of 20-30% on real problems!

`unique` may not be combined in the same patten with either `test` or `not` CEs.

`unique` was new in Jess 4.1, and is my own invention. I'm interested in hearing any feedback related to this feature.

2.16 Defglobals

Jess can support global variables that are visible from the command-prompt or inside any rule or deffunction. You can define them using the defglobal construct:

```
    (defglobal
         [<varname1> = <value1>]*)
```

Note that defglobals are reset to their assigned values by the (reset) command. If the <value> is a function call, this function will be evaluated each time (reset) is called. You can change this behaviour with the set-reset-globals command.

2.17 Things Not Implemented In Jess

Jess does not implement all features of all CLIPS constructs. This list tries to explain some of what's missing from Jess to those who know CLIPS. If you're not already a CLIPS user, you can skip this section.

2.17.1 Defrules

- The `and` and `or` conditional elements (CEs) are not supported on rule LHSs. `not` is supported, however. You can generally use multiple rules to simulate the effect of an `and` or `or` CE.
- The `|` connective constraint is not supported. Note that instead of writing a pattern like:

  ```
          (foo bar|baz)
  ```

  you can write:

  ```
          (foo ?x&:(or (eq ?X bar) (eq ?X baz)))
  ```

  to achieve the same effect in Jess.

2.17.2 Deffunctions.

Forward declarations of mutually recursive functions are not needed in Jess and will not parse.

2.17.3 Deftemplates.

The only supported slot attribute in Jess are the `default` and `default-dynamic` attributes. In particular, `type` will parse, but is ignored at runtime.

2.17.4 COOL, FuzzyCLIPS, wxCLIPS, etc.

Jess does not implement any features of these CLIPS extensions. Note that `defclass` and `definstance` are keywords in CLIPS that form part of COOL. Although these keywords exist in Jess, their syntax and precise meaning is different. You should find that the functionality they provide (pattern matching on Java Beans) is a satisfactory replacement for COOL.

2.17.5 Modules.

Jess does not implement CLIPS modules. However, since Jess itself is object-oriented, you can instantiate multiple Jess systems and get them to communicate via the external function interface.

---

3 Jess Function Guide

In this section, every Jess language function shipped with Jess version 4.3 is described. Some of these functions are intrinsic functions while others are Userfunctions and may not be available to all Jess code. All of these functions are installed into the command-line version of Jess; to use a function not marked (built-in) in your own programs, you need to add the appropriate Userpackage using `Rete.addUserpackage(new <pkgname>())`. The package for each function is listed below.

Note: many functions documented as requiring a specific minimum number of arguments will actually return sensible results with fewer; for example, the + function will return the value of a single argument as its result. This behavior is to be regarded as undocumented and unsupported. In addition, all functions documented as requiring a specific number of arguments will not report an error if invoked with more than that number; extra rguments are simply ignored.

`(* <numeric-expression> <numeric-expression>+)`

Package:
: (built-in)

Arguments:
: Two or more numeric expressions

Returns:
: Number

Description:

Returns the products of its arguments. The return value is an `INTEGER` unless any of the arguments are `FLOAT`, in which case it is a `FLOAT`.

(** <numeric-expression> <numeric-expression>)

Package:

`jess.MathFunctions`

Arguments:

Two numeric expressions

Returns:

Number

Description:

Raises its first argument to the power of its second argument (using Java's `Math.pow()` function). ***Note*:** the return value is `NaN` (not a number) if both arguments are negative.

(+ <numeric-expression> <numeric-expression>+)

Package:

(built-in)

Arguments:

Two or more numeric expressions

Returns:

Number

Description:

Returns the sum of its arguments. The return value is an `INTEGER` unless any of the arguments are `FLOAT`, in which case it is a `FLOAT`. ***Note***: the return value is the value of the single numeric expression if only one argument is supplied.

(- <numeric-expression> <numeric-expression>+)

Package:

(built-in)

Arguments:

Two or more numeric expressions

Returns:

Number

Description:

Returns the first argument minus all subsequent arguments. The return value is an `INTEGER` unless any of the arguments are `FLOAT`, in which case it is a `FLOAT`.

(/ <numeric-expression> <numeric-expression>+)

Package:

(built-in)

Arguments:

Two or more numeric expressions

Returns:

Number

Description:

Returns the first argument divided by all subsequent arguments. The return value is a `FLOAT`.

`(< <numeric-expression> <numeric-expression>+)`

Package:

(built-in)

Arguments:

Two or more numeric expressions

Returns:

Boolean

Description:

Returns `TRUE` if each argument is less in value than the argument following it; otherwise, returns `FALSE`.

`(<= <numeric-expression> <numeric-expression>+)`

Package:

(built-in)

Arguments:

Two or more numeric expressions

Returns:

Boolean

Description:

Returns `TRUE` if the value of each argument is less than or equal to the value of the argument following it; otherwise, returns `FALSE`.

`(<> <numeric-expression> <numeric-expression>+)`

Package:

(built-in)

Arguments:

Two or more numeric expressions

Returns:

Boolean

Description:

Returns `TRUE` if the value of the first argument is not equal in value to all subsequent arguments; otherwise returns `FALSE`.

`(= <numeric-expression> <numeric-expression>+)`

Package:

(built-in)

Arguments:

Two or more numeric expressions

Returns:

Boolean

Description:

Returns `TRUE` if the value of the first argument is equal in value to all subsequent arguments; otherwise, returns `FALSE`. The integer 2 and the float 2.0 are `=`, but not `eq`.

`(> <numeric-expression> <numeric-expression>+)`

Package:

(built-in)

Arguments:

Two or more numeric expressions

Returns:

Boolean

Description:

Returns `TRUE` if the value of each argument is less than that of the argument following it; otherwise, returns `FALSE`.

`(>= <numeric-expression> <numeric-expression>+)`

Package:

(built-in)

Arguments:

Two or more numeric expressions

Returns:

Boolean

Description:

Returns `TRUE` if the value of each argument is greater than or equal to that of the argument following it; otherwise, returns `FALSE`.

`(abs <numeric-expression>)`

Package:

`jess.MathFunctions`

Arguments:

One numeric expression

Returns:

Number

Description:

Returns the absolute value of its only argument.

`(agenda)`

Package:

jess.MiscFunctions

Arguments:

None

Returns:

NIL

Description:

Displays a list of rule activations to the WSTDOUT router.

`(and <expression>+)`

Package:
: (built-in)

Arguments:
: One or more expressions

Returns:
: Boolean

Description:

Returns `TRUE` if all arguments evaluate to a non-`FALSE` value; otherwise, returns `FALSE`.

`(assert <RHS-pattern>+)`

Package:
: (built-in)

Arguments:
: One or more facts (not fact-IDs)

Returns:
: Fact-ID or `FALSE`

Description:
: Adds a fact to the fact list. Asserts all facts onto the fact list; returns the fact-ID of last fact asserted or `FALSE` if no facts were successfully asserted (for example, if all facts given are duplicates of existing facts.)

`(assert-string <string-expression>)`

Package:
: (built-in)

Arguments:
: One string representing a fact

Returns:
: Fact-ID or `FALSE`

Description:
: Converts a string into a fact and asserts it. Attempts to parse string as a fact, and if successful, returns the value returned by assert with the same fact. Note that the string must contain the fact's enclosing parentheses.

`(bag <bag-command> <bag-arguments>+)`

Package:
: `jess.BagFunctions`

Arguments:
: An atom (a sub-command) and one or more additional arguments

Returns:
: (Varies)

Description:
: The `bag` command lets you manipulate Java hashtables from Jess. The net result is that you can create any number of

associative arrays or property lists. Each such array or list has a name by which it can be looked up. The lists can contain other lists as properties, or any other Jess data type.

The `bag` command does different things based on its first argument. It's really seven commands in one:

- `create` accepts a String, the name of a new Bag to be created. The bag object itself is returned. For example:

  ```
  Jess> (bag create my-bag) <External-Address> Jess>
  ```

- `delete` accepts the name of an existing bag, and deletes it from the list of bags.
- `find` accepts the name of a bag, and returns the corresponding bag object, if one exists, or `nil.`
- `list` returns a list of the names of all the existing bags, as a multifield.
- `set` accepts as arguments a bag, a String property name, and any Jess value as its three arguments. The named property of the given bag is set to the value, and the value is returned.
- `get` accepts as arguments a bag and a String property name. The named property is retrieved and returned, or `nil` if there is no such property. For example:

  ```
  Jess> (defglobal ?*bag* = 0)
  TRUE
  Jess> (bind ?*bag* (bag create my-bag))
  <External-Address>
  Jess> (bag set ?*bag* my-prop 3.0)
  3.0
  Jess> (bag get ?*bag* my-prop)
  3.0
  ```

- `props` accepts a bag as the single argument and returns a multifield consisting of a list of the names of all the properties of that bag.

`(batch)`

Package:

`jess.Miscfunctions`

Arguments:

One string or atom representing the name of a file

Returns:

(Varies)

Description:

Attempts to parse and evaluate the given file as Jess code. If successful, returns the return value of the last expression in the file.

Note: the argument must follow Jess' rules for valid atoms or strings. On UNIX systems, this presents no particular problems, but Win32 filenames may need special treatment. In particular: pathnames should use either '\\' (double backslash) or '/' (forward slash) instead of '\' (single backslash) as directory separators; and pathnames which include a colon (':') or a space character (' ') *must* be enclosed in double quotes.

`(bind <variable> <expression>*)`

Package:

(built-in)

Arguments:

A variable name and any value

Returns:

(Varies)

Description:

Binds a variable to a new value. Assigns the given value to the given variable, creating the variable if necessary. Note that (as in CLIPS) this works best in rules and deffunctions, and not from the command prompt. Returns the given value.

(build <lexeme-expression>)

Package:

`jess.Miscfunctions`

Arguments:

One string representing some Jess code

Returns:

(Varies)

Description:

Evaluates a string as though it were entered at the command prompt. Only allows constructs to be evaluated. Attempts to parse and evaluate the given string as Jess code. If successful, returns the return value of the last expression in the string. This is typically used to define rules from Jess code. For instance:

```
(build "(defrule foo (foo) => (bar))")
```

(call (<external-address> | <string-expression>) <string-expression> <call-arguments>+)

Package:

`jess.reflect.ReflectFunctions`

Arguments:

an external address or String, a String, and any number of additional arguments (see below)

Returns:

(Varies)

Description:

Calls a Java method on the given object, or a static method of the class named by the first argument. The second argument is the name of the method, and subsequent arguments are passed to the method. Arguments are promoted and overloaded methods selected precisely as for new. The return value is converted to a suitable Jess value before being returned. Array return values are converted to multifields.

The functor `call` may be omitted if the method being called is non-static and the object is represented by a simple variable. The following two method calls are equivalent:

```
        ;; These are legal and equivalent
        (call ?vector addElement (new java.lang.String "Foo"))
        (?vector addElement (new java.lang.String "Foo"))
```

`call` may not be omitted if the object comes from the return value of another function call:

```
        ;; This is illegal
        ((new java.lang.Vector 10) addElement (new java.lang.String "Foo"))
```

(clear)

Package:

(built-in)

Arguments:

None

Returns:

`TRUE`

Description:

Clears Jess. Deletes all rules, deffacts, defglobals, deftemplates, facts, activations, and so forth. Userfunctions are not deleted.

`(close [<router-identifier>])`

Package:

(built-in)

Arguments:

One or more router identifiers (atoms)

Returns:

`TRUE`

Description:

Closes any I/O routers associated with the given name by calling `close()` on the underlying stream, then removes the routers. Any subsequent attempt to use a closed router will report `bad router`. See open.

`(complement$ <multifield-expression> <multifield-expression>)`

Package:

`jess.MultiFunctions`

Arguments:

Two multifields

Returns:

Multifield

Description:

Returns a new multifield consisting of all elements of the second multifield not appearing in the first multifield.

`(create$ <expression>*)`

Package:

`jess.MultiFunctions`

Arguments:

Zero or more expressions

Returns:

Multifield

Description:

Appends its arguments together to create a multifield value. Returns a new multifield containing all the given arguments. ***Note***: multifields must be created explicitly using this function or others that return them. Multifields cannot be directly parsed from Jess input.

`(delete$ <multifield-expression> <begin-integer-expression> <end-integer-expression>)`

Package:

`jess.MultiFunctions`

Arguments:

A multifield and two integer expressions

Returns:

Multifield

Description:

Deletes the specified range from a multifield value. The first numeric expression is the 1-based index of the first element to remove; the second is the 1-based index of the last element to remove.

`(div <numeric-expression> <numeric-expression>+)`

Package:

`jess.MathFunctions`

Arguments:

Two or more numeric expressions

Returns:

Numbers

Description:

Returns the first argument divided by all subsequent arguments using integer division. Quotient of the values of the two numeric expressions rounded to the nearest integer.

`(e)`

Package:

`jess.MathFunctions`

Arguments:

None

Returns:

Number

Description:

Returns the transcendental number *e*.

`(engine)`

Package:

`jess.MiscFunctions`

Arguments:

None

Returns:

External address

Description:

Returns an external-address object containing the Rete engine in which the function in called.

`(eq <expression> <expression>+)`

Package:

(built-in)

Arguments:

Two or more arbitrary arguments

Returns:

Boolean

Description:

Returns `TRUE` if the first argument is equal in type and value to all subsequent arguments. For strings, this means identical contents. Uses the Java `Object.equals()` function, so can be redefined for external types. Note that the integer `2` and the floating-point number `2.0` are *not* `eq`, but they are `eq*` and `=`.

`(eq* <expression> <expression>+)`

Package:

(built-in)

Arguments:

Two or more arbitrary arguments

Returns:

Boolean

Description:

Returns `TRUE` if the first argument is equivalent to all the others. Uses numeric equality for numeric types, unlike `eq`. Note that the integer 2 and the floating-point number 2.0 are *not* `eq`, but they are `eq*` and =.

`(eval <lexeme-expression>)`

Package:

(built-in)

Arguments:

One string containing a valid Jess expression

Returns:

(Varies)

Description:

Evaluates a string as though it were entered at a command prompt. Only allows functions to be evaluated. Evaluates the string as if entered at the command line and returns the result.

`(evenp <expression>)`

Package:

`jess.PredFunctions`

Arguments:

One numeric expression

Returns:

Boolean

Description:

Returns `TRUE` for even numbers; otherwise, returns `FALSE`. Results with non-integers may be unpredictable.

`(exit)`

Package:

(built-in)

Arguments:

None

Returns:

Nothing

Description:

Exits Jess and halts Java.

`(exp <numeric-expression>)`

Package:

`jess.MathFunctions`

Arguments:

One numeric expression

Returns:

Number

Description:

Raises the value *e* to the power of its only argument.

`(explode$ <string-expression>)`

Package:

`jess.MultiFunctions`

Arguments:

One string

Returns:

Multifield

Description:

Creates a multifield value from a string. Parses the string as if by a succession of `read` calls, then returns these individual values as the elements of a multifield.

`(external-addressp <expression>)`

Package:

`jess.PredFunctions`

Arguments:

One expression

Returns:

Boolean

Description:

Returns TRUE or FALSE as the given expression is an external-address.

`(facts)`

Package:

(built-in)

Arguments:

None

Returns:

`TRUE`

Description:

Prints a list of all facts on the fact list.

`(fetch <string or atom>)`

Package:
: (built-in)

Arguments:
: One string or atom

Returns:
: (varies)

Description:
: Retrieves and returns any value previously stored by the `store` function under the given name, or `nil` if there is none. Analogous to the `fetch()` member function of the `Rete` class. See the section "Using store and fetch" for details.

`(first$ <multifield-expression>)`

Package:
: `jess.MultiFunctions`

Arguments:
: One multifield

Returns:
: Multifield

Description:
: Returns the first field of a multifield.as a new 1-element multifield.

`(float <numeric-expression>)`

Package:
: `jess.MathFunctions`

Arguments:
: One numeric expression

Returns:
: Floating-point number

Description:
: Converts its only argument to a float.

`(floatp <expression>)`

Package:
: `jess.PredFunctions`

Arguments:
: One numeric expression

Returns:
: Boolean

Description:
: Returns TRUE for floats; otherwise, returns FALSE.

`(foreach <variable> <multifield-expression> <action>*)`

Package:
: (built-in)

Arguments:
: A variable, a multifield expression, and zero or more arguments

Returns:
: Varies

Description:
: The named variable is set to each of the values in the multifield in turn; for each value, all of the other arguments are evaluated in order. The `return` function can be used to break the iteration.

  Example:

  ```
  (foreach ?x (create$ a b c d) (printout t ?x crlf))
  ```

`(format <router-identifier> <string-expression> <expression>*)`

Package:
: `jess.MiscFunctions`

Arguments:
: A router identifier, a format string, and zero or more arguments

Returns:
: A string

Description:
: Sends formatted output to the specified logical name. Formats the arguments into a string according to the format string, which is identical to that used by `printf` in the C language (find a C book for more information). Returns the string, and optionally prints the string to the named router. If you pass `nil` for the router name, no printing is done.

`(get <external-address> <string-expression>)`

Package:
: `jess.reflect.ReflectFunctions`

Arguments:
: An external address and a string.

Returns:
: (Varies)

Description:
: Retrieves the value of a Java Bean's property. The first argument is the object and the second argument is the name of the property. The return value is converted to a suitable Jess value exactly as for call.

`(get-member (<external-address> | <string-expression>) <string-expression>)`

Package:
: `jess.reflect,ReflectFunctions`

Arguments:
: An external address or a string, and a string.

Returns:
: (Varies)

Description:

Retrieves the value of a Java object's data member. The first argument is the object (or the name of a class, for a static member) and the second argument is the name of the field. The return value is converted to a suitable Jess value exactly as for call.

`(get-reset-globals)`

Package:
: jess.MiscFunctions

Arguments:
: None

Returns:
: Boolean

Description:
: Indicates the current setting of global variable reset behavior. See set-reset-globals for an explanation of this property.

`get-salience-evaluation`

Package:
: jess.MiscFunctions

Arguments:
: None

Returns:
: Atom

Description:
: Indicates the current setting of salience evaluation behavior. See set-salience-evaluation for an explanation of this property.

`(gensym*)`

Package:
: (built-in)

Arguments:
: None

Returns:
: Atom

Description:
: Returns an atom which consists of the letters `gen` plus an integer. Use `setgen` to set the value of the integer to be used by the next gensym call. Note that, unlike in CLIPS, these symbols are *not* guaranteed to be unique. This will change in a future release.

`(get-var <lexeme-expression>)`

Package:
: (built-in)

Arguments:
: A string or atom

Returns:
: (Varies)

Description:

Fetches the value of a variable, given the name of the variable as a string or atom. (Rarely needed, but when you need it, you'll know.) Most commonly, get-var is used to fetch the value of a global variable on the LHS of a rule.

(halt)

Package:

(built-in)

Arguments:

None

Returns:

`TRUE`

Description:

Halts rule execution. No effect unless called from the RHS of a rule.

(if <expression> then <action>* [else <action>*])

Package:

(built-in)

Arguments:

A Boolean variable or function call returning Boolean, the atom `then`, and any number of additional expressions; optionally followed by the atom `else` another list of expression.

Returns:

(Varies)

Description:

Allows conditional execution of a group of actions. The boolean expression is evaluated. If it does not evaluate to `FALSE`, the first list of expressions is evaluated, and the return value is that returned by the last expression of that list. If it does evaluate to `FALSE`, and the optional second list of expressions is supplied, those expressions are evaluated and the value of the last is returned.

Example:

```
(if (> ?x 100)
        then
          (printout t "X is big" crlf)
        else
          (printout t "X is small" crlf))
```

(implode$ <multifield-expression>)

Package:

`jess.MultiFunctions`

Arguments:

One multifield

Returns:

String

Description:

Creates a string from a multifield value. Converts each element of the multifield to a string, and returns these strings concatenated with single intervening spaces.

`(insert$ <multifield-expression> <integer-expression> <single-or-multifield-expression>+)`

Package:
: `jess.MultiFunctions`

Arguments:
: A multifield, an integer, and one more more multifields

Returns:
: A multifield

Description:
: Inserts one or more values in a multifield. Inserts the elements of the one or more multifields so that they appear starting at the given 1-based index of the first multifield.

`(integer <numeric-expression>)`

Package:
: `jess.MathFunctions`

Arguments:
: One numeric expression

Returns:
: Integer

Description:
: Converts its only argument to an integer. Truncates any fractional component of the value of the given numeric expression and returns the integral part.

`(integerp <expression>)`

Package:
: `jess.PredFunctions`

Arguments:
: One expression

Returns:
: Boolean

Description:
: Returns `TRUE` for integers; otherwise, returns `FALSE`.

`(intersection$ <multifield-expression> <multifield-expression>)`

Package:
: `jess.MultiFunctions`

Arguments:
: Two multifields

Returns:
: Multifield

Description:
: Returns the intersection of two multifields. Returns a multifield consisting of the elements the two argument multifields have in common.

`(jess-version-number)`

Package:
: (built-in)

Arguments:
: None

Returns:
: Float

Description:
: Returns a version number for Jess; currently 4.3 .

`(jess-version-string)`

Package:
: (built-in)

Arguments:
: None

Returns:
: String

Description:
: Returns a human-readable string descriptive of this version of Jess.

`(length$ <multifield-expression>)`

Package:
: `jess.MultiFunctions`

Arguments:
: Multifield

Returns:
: Integer

Description:
: Returns the number of fields in a multifield value.

`(lexemep <expression>)`

Package:
: `jess.PredFunctions`

Arguments:
: Any expression

Returns:
: Boolean

Description:
: Returns `TRUE` for symbols and strings; otherwise, returns `FALSE`.

`(list-function$)`

Package:

(built-in)

Arguments:

None

Returns:

Multifield

Description:

Returns a multifield list of all the functions currently callable, including intrinsics, deffunctions, and Userfunctions. Each function name is an atom. The names are sorted in alphabetical order.

`(load-facts <file-name>)`

Package:

(built-in)

Arguments:

A string or atom representing the name of a file of facts

Returns:

Boolean

Description:

Asserts facts loaded from a file. The argument should name a file containing a list of facts (not deffacts constructs, and no other commands or constructs). Jess will parse the file and assert each fact. The return value is the return value of assert when asserting the last fact. In an applet, `load-facts` will use `getDocumentBase()` to find the named file.

Note: See the batch command for a discussion about specifying filenames in Jess.

`(load-function <class-name>)`

Package:

`jess.MiscFunctions`

Arguments:

One string or atom representing the name of a Java class

Returns:

Boolean

Description:

The argument must be the fully-qualified name of a Java class that implements the Userfunction interface. The class is loaded in to Jess and added to the engine, thus making the corresponding command available. See Extending Jess with Java for more information.

`(load-package <class-name>)`

Package:

`jess.MiscFunctions`

Arguments:

One string or atom, the name of a Java class

Returns:

Boolean

Description:

The argument must be the fully-qualified name of a Java class that implements the Userpackage interface. The class is loaded in to Jess and added to the engine, thus making the corresponding package of commands available. See Extending Jess with Java for more information.

`(log <numeric-expression>)`

Package:
: `jess.MathFunctions`

Arguments:
: One numeric expression

Returns:
: Number

Description:
: Returns the logarithm base `e` of its only argument.

`(log10 <numeric-expression>)`

Package:
: `jess.MathFunctions`

Arguments:
: One numeric expression

Returns:
: Number

Description:
: Returns the logarithm base-10 of its only argument.

`(lowcase <lexeme-expression>)`

Package:
: `jess.StringFunctions`

Arguments:
: One atom or string.

Returns:
: String

Description:
: Converts uppercase characters in a string or symbol to lowercase. Returns the argument as an all-lowercase string.

`(max <numeric-expression>+)`

Package:
: `jess.MathFunctions`

Arguments:
: One or more numerical expressions

Returns:
: Number

Description:
: Returns the value of its largest numeric argument

`(member$ <single-field-expression> <multifield-expression>)`

Package:
: `jess.MultiFunctions`

Arguments:
: A value and a multifield

Returns:
: Integer or `FALSE`

Description:
: Returns the position (1-based index) of a single-field value within a multifield value; otherwise, returns `FALSE`.

`(min <numeric-expression>+)`

Package:
: `jess.MathFunctions`

Arguments:
: One or more numeric expressions

Returns:
: Number

Description:
: Returns the value of its smallest numeric argument.

`(mod <numeric-expression> <numeric-expression>)`

Package:
: (built-in)

Arguments:
: Two integer expressions

Returns:
: Integer

Description:
: Returns the remainder of the result of dividing the first argument by its second (assuming that the result of the division must be an integer).

`(modify <fact-specified> <RHS-slot>*)`

Package:
: (built-in)

Arguments:
: A fact-ID and zero or more two-element lists

Returns:
: Fact-ID

Description:
: Modifies the deftemplate fact in the fact list. The fact-ID must belong to an unordered fact. Each list is taken as the name of a slot in this fact and a new value to assign to the slot. A new fact is asserted which is similar to the given fact but which has the specified slots replaced with new values. The original fact is retracted. The fact-ID of the new fact is returned. Modifying a `definstance` fact will cause the appropriate object properties to be set as well.

`(multifieldp <expression>)`

Package:
: `jess.PredFunctions`

Arguments:
: Any value

Returns:
: Boolean

Description:
: Returns `TRUE` for multifield values; otherwise, returns `FALSE`.

`(neq <expression> <expression>+)`

Package:
: (built-in)

Arguments:
: Two or more values

Returns:
: Boolean

Description:
: Returns `TRUE` if the first argument is not equal in type and value to all subsequent arguments (see `eq`).

`(new <string-expression> <new-arguments>+)`

Package:
: `jess.reflect.ReflectFunctions`

Arguments:
: A string and one or more arguments

Returns:
: Boolean

Description:
: Creates a new Java object and returns an `EXTERNAL_ADDRESS` value containing it. The first argument is the fully-qualified class name: `java.util.Vector`, for example. The second and later arguments are constructor arguments. The constructor will be chosen from among all constuctors for the named class based on a *first-best fit* algorithm. Built-in Jess types are converted as necessary to match available constructors. See the text for more details.

`(not <expression>)`

Package:
: (built-in)

Arguments:
: One expression

Returns:
: Boolean

Description:
: Returns `TRUE` if its only arguments evaluates to `FALSE`; otherwise, returns `FALSE`.

`(nth$ <integer-expression> <multifield-expression>)`

Package:
: `jess.MultiFunctions`

Arguments:
: A number and a multifield

Returns:
: (Varies)

Description:
: Returns the value of the specified (1-based index) field of a multifield value.

`(numberp <expression>)`

Package:
: `jess.PredFunctions`

Arguments:
: One expression

Returns:
: Boolean

Description:
: Returns `TRUE` for numbers; otherwise, returns `FALSE`.

`(oddp <integer-expression>)`

Package:
: `jess.PredFunctions`

Arguments:
: One integer expression

Returns:
: Boolean

Description:
: Returns `TRUE` for odd numbers; otherwise, returns `FALSE`; see `evenp`.

`(open <file-name> <router-identifier> ["r"|"w"|"a"])`

Package:
: (built-in)

Arguments:
: A file name, an identifier for the file (an atom), and optionally a mode string: one of `r`, `w`, `a`.

Returns:
: The file identifier

Description:
: Opens a file. Subsequently, the given router identifier can be passed to `printout`, `read`, `readline`, or any other functions that accept I/O routers as arguments. By default, the file is opened for reading; if a mode string is given, it may be opened for reading only (`r`), writing only (`w`), or appending (`a`).

: ***Note:*** See the batch command for a discussion about specifying filenames in Jess.

`(or <expression>+)`

Package:

(built-in)

Arguments:

One or more expressions

Returns:

Boolean

Description:

Returns `TRUE` if any of the arguments evaluates to a non-`FALSE` value; otherwise, returns `FALSE`.

`(pi)`

Package:

`jess.MathFunctions`

Arguments:

None

Returns:

Number

Description:

Returns the number `pi`.

`(ppdefrule <rule-name>)`

Package:

`jess.MiscFunctions`

Arguments:

A string or atom representing the name of a rule

Returns:

String containing rule's text

Description:

Displays the text of a given rule in a pretty-print representation.

`(printout <router-identifier> <expression>*)`

Package:

(built-in)

Arguments:

A router identifier followed by zero or more expressions

Returns:

`nil`

Description:

Sends unformatted output to the specified logical name. Prints its arguments to the named router, which must be open for output. No spaces are added between arguments. The special atom `crlf` prints as a newline. The special router name `t` can be used to signify standard output.

`(random)`

Package:

`jess.MathFunctions`

Arguments:

None

Returns:

Number

Description:

Returns a pseudo-random integer between 0 and 65536.

(read [<router-identifier>])

Package:

(built-in)

Arguments:

An optional input router identifier (when omitted `t` is the default)

Returns:

(Varies)

Description:

Reads a single-field value from a specified logical name. Read a single atom, string, or number from the named router, returns this value. The router `t` means standard input. Newlines are treated as ordinary whitespace; this behaviour is different than in CLIPS, which returns newlines as tokens. If you need to parse text line-by-line, use `readline` and `explode$`.

(readline [<router-identifier>])

Package:

(built-in)

Arguments:

An optional input router identifier (when omitted `t` is the default)

Returns:

String

Description:

Reads an entire line as a string from the specified logical name (router). The router `t` means standard input.

(replace$ <multifield-expression> <begin-integer-expression> <end-integer-expression> <single-or-multifield-expression>+)

Package:

`jess.MultiFunctions`

Arguments:

A multifield, two numeric expressions, and one or more multifields

Returns:

Multifield

Description:

Replaces the specified range of a multifield value with a set of values. The last one more more multifields are inserted into the first multifield, replacing elements between the 1-based indices given by the two numeric arguments, inclusive.

Example:

```
Jess> (replace$ (create$ a b c) 2 2 (create$ x y z))
    (a x y z c)
```

`(reset)`

Package:
: (built-in)

Arguments:
: None

Returns:
: `TRUE`

Description:
: Removes all facts from the fact list, removes all activations, then asserts the fact `(initial-fact)`, then asserts all facts found in deffacts, asserts a fact representing each registered definstance, and (if the set-reset-globals property is TRUE) initializes all defglobals.

`(rest$ <multifield-expression>)`

Package:
: `jess.MultiFunctions`

Arguments:
: One multifield

Returns:
: Multifield

Description:
: Returns all but the first field of a multifield as a new multifield.

`(retract <integer-expression>+)`

Package:
: (built-in)

Arguments:
: One or more fact-IDs

Returns:
: `TRUE`

Description:
: Retracts the facts whose IDs are given. Retracting a definstance fact will result in an implict call to undefinstance for the corresponding object (the object will no longer be pattern-matched).

`(return [<expression>])`

Package:
: (built-in)

Arguments:
: An optional expression

Returns:
: (Varies)

Description:

Returns the given value from a deffunction. Exits the deffunction immediately.

`(round <numeric-expression>)`

Package:
: `jess.MathFunctions`

Arguments:
: One numeric expression

Returns:
: Integer

Description:
: Rounds its argument toward the closest integer or negative infinity if exactly between two integers.

`(rules)`

Package:
: (built-in)

Arguments:
: None

Returns:
: `TRUE`

Description:
: Prints a list of all defrules.

`(run)`

Package:
: (built-in)

Arguments:
: None

Returns:
: `TRUE`

Description:
: Starts the inference engine. Jess will keep running until no more activations remain or `halt` is called.

`(save-facts <file-name> [<deftemplate-name>])`

Package:
: (built-in)

Arguments:
: A filename, and optionally an atom

Returns:
: Boolean

Description:
: Saves facts to a file. Attempts to open the named file for writing, and then writes a list of all facts on the fact list to the file. This file is suitable for reading with load-facts. If the optional second argument is given, only facts whose head matches this atom will be saved. Does not work in applets.

Note: See the batch command for a discussion about specifying filenames in Jess.

`(set <external-address> <string-expression> <expression>)`

Package:

`jess.reflect.ReflectFunctions`

Arguments:

An external address, a string, and an expression

Returns:

The last argument

Description:

Sets a Java Bean's property to the given value. The first argument is the Bean object; the second argument is the name of the property. The third value is the new value for the property; the same conversions are applied as for `new` and `call`.

`(set-member (<external-address> | <string-expression>) <string> <expression>+)`

Package: `jess.reflect.ReflectFunctions`

Arguments:

An external address or a string, a string, and one or more expressions

Returns:

The last argument

Description:

Sets a Java object's member variable to the given value. The first argument is the object (or the name of the class, in the case of a static member variable). The second argument is the name of the variable. The third value is the new value for the variable; the same conversions are applied as for `new` and `call`.

`(set-reset-globals (TRUE | FALSE | nil))`

Package:

jess.MiscFunctions

Arguments:

One boolean value (TRUE or FALSE or nil)

Returns:

Boolean

Description:

Changes the current setting of the global variable reset behavior. If this property is set to TRUE (the default), then the (reset) command reinitializes the values of global variables to their initial values (if the initial value was a function call, the function call is reexecuted.) If the property is set to FALSE or nil, then (reset) will not affect global variables. Note that in previous versions of Jess, defglobals were always reset; but if the initial value was set with a fucntion call, the function was **not** reevaluated. Now it is.

`(set-salience-evaluation (when-defined | when-activated | every-cycle))`

Package:

jess.MiscFunctions

Arguments:

One of the atoms `when-defined`, `when-activated`, or `every-cycle`

Returns:

One of the potential arguments (the previous value of this property)

Description:

Changes the current setting of the salience evaluation behavior. By default, a rule's salience will be determined once, when the rule is defined (when-defined.) If this property is set to when-activated, then the salience of each rule will be redetermined immediately before each time it is placed on the agenda. If the property is set to every-cycle, then the salience of every rule is redetermined immediately after each time any rule fires.

`(set-strategy (depth | breadth))`

Package:

`jess.MiscFunctions`

Arguments:

An atom or string representing the name of a strategy (can be a fully-qualified Java class name). You can use `depth` and `breadth` to represent the two built-in strategies.

Returns:

The previous strategy as an atom

Description:

Lets you specify the *conflict resolution strategy* Jess uses to order the firing of rules of equal salience. Currently, there are two strategies available: *depth (LIFO)* and *breadth (FIFO)*. When the depth strategy is in effect (the default), more recently activated rules are fired before less recently activated rules of the same salience. When the breadth strategy is active, rules of the same salience fire in the order in which they are activated. Note that in either case, if several rules are activated simultaneously (i.e., by the same fact-assertion event) the order in which these several rules fire is unspecified, implementation-dependent and subject to change. More built-in strategies may be added in the future. You can (perhaps) implement your own strategies in Java by creating a class that implements the `jess.Strategy` interface and then specifying its fully-qualified classname as the argument to `set-strategy`. Details can be gleaned from the source. At this time, though, I think some of the methods you'd need to call are package-protected.

`(setgen <numeric-expression>)`

Package:

`jess.MiscFunctions`

Arguments:

A numeric expression

Returns:

`TRUE`

Description:

Sets the starting number used by `gensym*`. Note that if this number has already been used, `gensym*` uses the next larger number that has not been used.

`(socket <Internet-hostname> <TCP-port-number> <router-identifier>)`

Package:

`jess.MiscFunctions`

Arguments:

An Internet hostname, a TCP port number, and a router identifier

Returns:

The router identifier

Description:

Somewhat equivalent to `open`, except that instead of opening a file, opens an unbuffered TCP network connection to the named host at the named port, and installs it as a pair of read and write routers under the given name.

`(sqrt <numeric-expression>)`

Package:
: `jess.MathFunctions`

Arguments:
: A numeric expression

Returns:
: Number

Description:
: Returns the square root of its only argument.

`(store <string or atom> <expression>)`

Package:
: `(built-in)`

Arguments:
: A string or atom and any other value

Returns:
: (varies)

Description:
: Associates the expression with the name given by the first argument, such that later calls to the fetch will retrieve it. Analagous to the `store()` member function of the `jess.Rete class. See section on Using store and fetch for more details.`

`(str-cat <expression>*)`

Package:
: `jess.StringFunctions`

Arguments:
: Zero or more expressions

Returns:
: String

Description:
: Concatenates its arguments as strings to form a single string.

`(str-compare <string-expression> <string-expression>)`

Package:
: `jess.StringFunctions`

Arguments:
: Two strings

Returns:
: Integer

Description:
: Lexicographically compares two strings. Returns 0 if the strings are identical, a negative integer if the first is lexicographically less than the second, a positive integer if lexicographically greater.

(str-index <lexeme-expression> <lexeme-expression>)

Package:
: `jess.StringFunctions`

Arguments:
: Two atoms

Returns:
: Integer or `FALSE`

Description:
: Returns the position of the first argument within the second argument. This is the 1-based index at which the first string first appears in the second; otherwise, returns `FALSE`.

(str-length <lexeme-expression>)

Package:
: `jess.StringFunctions`

Arguments:
: An atom

Returns:
: Integer

Description:
: Returns the length of an atom in characters.

(stringp <expression>)

Package:
: `jess.PredFunctions`

Arguments:
: One expression

Returns:
: Boolean

Description:
: Returns `TRUE` for strings; otherwise, returns `FALSE`.

(sub-string <begin-integer-expression> <end-integer-expression> <string-expression>)

Package:
: `jess.StringFunctions`

Arguments:
: Two numbers and a string

Returns:
: String

Description:
: Retrieves a subportion from a string. Returns the string consisting of the characters between the two 1-based indices of the given string, inclusive.

(subseq$ <multifield-expression> <begin-integer-expression>

<end-integer-expression>)

Package:
: `jess.MultiFunctions`

Arguments:
: A multifield and two numeric expressions

Returns:
: Multifield

Description:
: Extracts the specified range from a multifield value consisting of the elements between the two 1-based indices of the given multifield, inclusive.

(subsetp <multifield-expression> <multifield-expression>)

Package:
: `jess.MultiFunctions`

Arguments:
: Two multifields

Returns:
: Boolean

Description:
: Returns `TRUE` if the first argument is a subset of the second (i.e., all the elements of the first multifield appear in the second multifield); otherwise, `returns FALSE`.

(sym-cat <expression>*)

Package:
: (built-in)

Arguments:
: Zero or more expressions

Returns:
: Atom

Description:
: Concatenates its arguments as strings to form a single symbol.

(symbolp <expression>)

Package:
: `jess.PredFunctions`

Arguments:
: One expression

Returns:
: Boolean

Description:
: Returns `TRUE` for symbols; otherwise, returns `FALSE`.

(system <lexeme-expression>*)

Package:

`jess.MiscFunctions`

Arguments:

Zero or more atoms

Returns:

`TRUE`

Description:

Appends its arguments to form a command which is then sent to the operating system. Executes the operating-system command-line constructed by converting each argument to a string. Normally blocks (i.e., Jess stops until the applet returns), but if the last argument is an ampersand (&), the program will run in the background.

`(time)`

Package:

`jess.MiscFunctions`

Arguments:

None

Returns:

Number

Description:

Returns the number of seconds since 12:00 AM, Jan 1, 1970.

`(try <expression>* catch <expression>*)`

Package:

(built-in)

Arguments:

One or more expressions, followed by the atom `catch`, followed by zero or more expressions

Returns:

(Varies)

Description:

This command works something like Java `try` with a few simplifications. The biggest difference is that the `catch` clause can specify neither a type of exception nor a variable to receive the exception object. All exceptions occurring in a `try` block are routed to the single required `catch` block. The variable `?ERROR` is made to point to the exception object as an `EXTERNAL_ADDRESS`. For example:

```
(try
(open NoSuchFile.txt r)
catch
(printout t (call ?ERROR toString) crlf))
```

prints

```
Rete Exception in routine _open::call.
Message: I/O Exception java.io.FileNotFoundException: NoSuchFile.txt.
```

An empty `catch` block is fine. It just signifies ignoring possible errors.

`(undefrule <rule-name>)`

Package:
: (built-in)

Arguments:
: An atom representing the name of a rule

Returns:
: Boolean

Description:
: Deletes a defrule. Removes the named rule from the Rete network and returns `TRUE` if the rule existed.. This rule will never fire again.

`(union$ <multifield-expression> <multifield-expression>)`

Package:
: `jess.MultiFunctions`

Arguments:
: Two multifields

Returns:
: Multifield

Description:
: Returns a new multifield consisting of the union of its two multifield arguments (i.e., of all the elements that appear in the two arguments with duplicates removed).

`(unwatch <watch-item>)`

Package:
: (built-in)

Arguments:
: One of the atoms `all`, `rules`, `compilations`, `activations`, `facts`

Returns:
: `TRUE`

Description:
: Causes trace output to not be printed for the given indicator. See `watch`.

`(upcase <lexeme-expression>)`

Package:
: `jess.StringFunctions`

Arguments:
: A string or atom

Returns:
: A string

Description:
: Converts lowercase characters in a string or symbol to uppercase. Returns the argument as an all-uppercase string.

`(view)`

Package:
: `jess.view.ViewFunctions`

Arguments:

None

Returns:

`TRUE`

Description:

This Userfunction is included in the Jess distribution but is not normally installed. It requires Java 1.1. You must load it using `load-package` (the class name is `jess.view.ViewFunctions`). When invoked, it displays a live snapshot of the Rete network in a graphical window. See How Jess Works.

`(watch (all | rules | compilations | activations | facts))`

Package:

(built-in)

Arguments:

One of the atoms `all`, `rules`, `compilations`, `activations`, `facts`

Returns:

`TRUE`

Description:

Produces additional debug output when specific events happen in Jess, depending on the argument. Any number of different watches can be active simultaneously:

- `rules`: prints a message when any rule fires.
- `compilations`: prints a message when any rule is compiled.
- `activations`: prints a message when any rule is activated, or deactivated, showing which facts have caused the event.
- `facts`: print a message whenever a fact is asserted or retracted.
- `all`: all of the above.

`(while <expression> [do] <action>*)`

Package:

(built-in)

Arguments:

A Boolean value or a function call returning Boolean, the atom `do`, and zero or more expressions

Returns:

(Varies)

Description:

Allows conditional looping. Evaluates the boolean expression repeatedly. As long as it does not equal `FALSE`, the list of other expressions are evaluated. The value of the last expression evaluated is the return value.

---

4 Writing Jess Code

Many useful expert systems can be written using only the Jess language as presented above. I won't present a tutorial on writing such systems here (maybe someday!), but I do want to share a few useful hints and ideas.

4.1 Using an External Editor

Jess allows you to enter rules directly at its interactive prompt. While this is fine for experimenting, Jess doesn't yet have the ability to remember the source text for all the rules and constructs you enter. Therefore, you will typically enter your rules and other data into a separate script file and read it into Jess using the `batch` command. Jess does offer the `ppdefrule` and `save-facts` commands, both of which can be very helpful in interactively building up a system definition and them storing it in a file. And as described in a previous section, you can use the `system` command to start the external editor form within Jess, if desired.

4.2 Efficiency

The single biggest determinant of Jess performance is the number of *partial matches* generated by your rules. You should always try to obey the following (sometimes contradictory) guidelines while writing your rules:

- Put the *most specific* patterns (those that will match the fewest facts) near the top of each rule's LHS.
- Put the *most transient* patterns (those that will match facts that are frequently retracted and asserted) near the bottom of a LHS.

You can use the `view` command to find out how many partial matches your rules generate. See How Jess Works.

4.3 Error Reporting and Debugging

I've tried hard to improve Jess's syntax error reporting in this release, but it is still not as detailed as it could be. When you get an error from Jess (during parsing or at runtime) it is generally delivered as a Java exception. The exception will contain an explanation of the problem and the stack trace of the exception will help you understand what went wrong. For this reason, it is *very important* that, if you're embedding Jess in a Java application, you don't write code like this:

```
        try
        {
          Rete engine;
          ...
          engine.executeCommand("(gibberish!)");
        }
        catch (ReteException re) { /* ignore errors */ }
```

If you ignore the Java exceptions, you will miss Jess's explanations of what's wrong with your code. Don't laugh - more people code this way than you'd think!

Anyway, if you attempt to load the folowing rule in the standard Jess command-line executable,

```
        Jess> (defrule foo-1
                (foo bar)
                ->
                (printout "Found Foo Bar" crlf))
```

You'll get the following printout:

```
        Rete Exception in routine Jesp::parseDefrule.
          Message: Expected '=>' at line 2:  ( defrule foo-1 ( foo bar ) -> .
                at jess.Jesp.parseError(Compiled Code)
                at jess.Jesp.doParseDefrule(Compiled Code)
                at jess.Jesp.parseDefrule(Compiled Code)
                at jess.Jesp.parseSexp(Compiled Code)
                at jess.Jesp.parse(Compiled Code)
                at jess.Main.main(Compiled Code)
```

Looking at the routine names listed in the stack trace make it fairly clear that a `defrule` was being parsed, and the detail message explains that the position of the `.` was reached in the input without finding the expected `=>` symbol (we accidentally typed `->` instead).

Runtime errors can be more puzzling, but the stack trace will give you a lot of information. Here's a rule where we erroneously try to add the number `3.0` to the word `four`:

```
        Jess> (defrule foo-2
                    =>
                      (printout t (+ 3.0 four) crlf))
```

When we `(reset)` and `(run)` we'll see:

```

       Rete Exception in routine Value::intValue while executing defrule foo-2.
          Message: Not a number: four type = 1 at line 8:  ( run ) .
                at jess.Value.typeError(Compiled Code)
                at jess.Value.numericValue(Compiled Code)
                at jess.Plus.call(Compiled Code)
                at jess.Funcall.simpleExecute(Compiled Code)
                at jess.Funcall.execute(Compiled Code)
                at jess.Funcall.execute(Compiled Code)
                at jess.Funcall.execute(Compiled Code)
                at jess.Defrule.fire(Compiled Code)
                at jess.Activation.fire(Compiled Code)
                at jess.Rete.run(Compiled Code)
                at jess.Rete.run(Compiled Code)
                at jess.HaltEtc.call(Compiled Code)
                at jess.Funcall.simpleExecute(Compiled Code)
                at jess.Funcall.execute(Compiled Code)
                at jess.Funcall.execute(Compiled Code)
                at jess.Jesp.parseAndExecuteFuncall(Compiled Code)
                at jess.Jesp.parseSexp(Compiled Code)
                at jess.Jesp.parse(Compiled Code)
                at jess.Main.main(Compiled Code)
```

In this case, the error message is pretty clear except for the claim that the offending statement is `run`. To find out what was really happening, we have to look at the stack trace. Starting from the top down, we find `Value.numericValue()` was called by `Plus.call()`. A few levels down, we see `Defrule.fire().` Taken together, this means that an addition operation on the RHS of the rule `foo-2` (from the first line of the trace) found the symbol `four` as one of its operands when it expected a number.

The notation `type = 1` in the error message, by the way, refers to a set of constants in the class `jess.RU.` The values of these constants are presented in section 5.4.1, The class jess.Value. Consulting that table, we see that type `1` is `RU.ATOM`, a symbol, which is indeed not a number.

If we make a similar mistake on the LHS of a rule:

```
        Jess> (defrule foo-3
                (test (eq 3 (+ 2 one)))
                 =>
                )
```

We see the following after a `reset`:

```

        Rete Exception in routine Value::intValue while executing 'test' CE:
        [NodeTest ntests=1 [Test1: test=EQ;slot_idx=3;sub_idx=-1;
                  slot_value=eq 3 + 2 one] ;usecount = 1].
            Message: Not a number: one type = 1 at line 11: ( reset ) .
                at jess.Value.typeError(Compiled Code)
                at jess.Value.numericValue(Compiled Code)
                at jess.Plus.call(Compiled Code)
                at jess.Funcall.simpleExecute(Compiled Code)
                at jess.Funcall.execute(Compiled Code)
                at jess.Funcall.execute(Compiled Code)
                at jess.Funcall.execute(Compiled Code)
                at jess.NodeTest.runTests(Compiled Code)
                at jess.NodeTest.callNode(Compiled Code)
                at jess.Node.passAlong(Compiled Code)
                at jess.Node1TELN.callNode(Compiled Code)
                at jess.Node.passAlong(Compiled Code)
                at jess.Node1TECT.callNode(Compiled Code)
                at jess.Rete.processTokenOneNode(Compiled Code)
                at jess.Rete.updateNodes(Compiled Code)
                at jess.ReteCompiler.addRule(Compiled Code)
                at jess.Rete.addDefrule(Compiled Code)
                at jess.Jesp.doParseDefrule(Compiled Code)
                at jess.Jesp.parseDefrule(Compiled Code)
                at jess.Jesp.parseSexp(Compiled Code)
                at jess.Jesp.parse(Compiled Code)
                at jess.Main.main(Compiled Code)
```

Again, the error message is somewhat but not completely helpful (it says the error was in the function `reset`, but it also says that a (test) CE was being executed, and it prints out a stylized version of the test CE itself) and the stack trace contains additional information. Here we see our old friends `Value.numericValue()` and `Plus.call()` being called, but we don't see the Defrule being fired. Instead we see lots of oddly named classes and functions with names containing `Node` and `Token`. This is always a tip-off that the error happened on Defrule LHS processing, as are both the fact that the error happened during a reset and the message at the top of the trace. Way down the stack we see `Rete.assert()` being called by `Rete.reset()`, which also indicates that LHS processing was in progress when the exception happened.

The funny string starting with `[NodeTest ntests=1; ...` is Jess's internal representation for a single node in the Rete network that encodes the `test` CE from the rule above. Looking at it, you can see that it includes the function call `(+ 2 one)`, which should help you track it down. Note that in this case, Jess *can't* tell you which rule this node belongs to, as it theoretically could be shared by several rules (see How Jess Works for details.)

---

5 Embedding Jess in a Java Program

There are three different ways to use Jess and Java code together:

- You can embed Jess in your own Java applications. This is covered in this Section of the manual.
- You can write Java classes which you can add to Jess so that they will appear as a part of the Jess language. This is discussed in Extending Jess With Commands Written in Java.
- You can manipulate Java objects directly from the Jess language using the optional `jess.reflect` package (the functions `new`, `call`, `set`, `get`, `set-member`, `get-member`, `defclass`, and `definstance`). This is covered in Accessing Java Objects Directly From Jess.

5.1 The `jess.Rete` Class

5.1.1 Executing a File of Jess Code

Using Jess from Java code is simple. The `jess.Rete` class contains the expert system engine. The `jess.Jesp` class contains the Jess parser. To execute a file of CLIPS code in Jess (like the Jess `batch` command), simply create a `Rete` object and a `Jesp` object, tell the `Jesp` object about the file, and call `Jesp.parse(boolean prompt)`:

```
import jess.*;

  // ...
  // See info about the ReteDisplay classes below
  NullDisplay nd = new NullDisplay();

  // Create a Jess engine
  Rete rete = new Rete(nd);

  // Open the file test.clp
  FileInputStream fis = new FileInputStream("test.clp");

  // Create a parser for the file, telling it where to take input
  // from and which engine to send the results to
  Jesp j = new Jesp(fis, rete);
  do
    {
      try
      {
        // parse and execute one construct, without printing a prompt
        j.parse(false);
      }
      catch (ReteException re)
      {
        // All Jess errors are reported as 'ReteException's.
        re.printStackTrace(nd.stderr());
      }
    } while (fis.available() > 0);
```

Note that if the file `test.clp` contains the CLIPS `reset` and `run` commands, the Jess engine will run to completion during the `parse()` calls. Also note that Jess will throw `jess.ReteException` exceptions to signal errors.

5.1.2 Adding Optional Commands

The code above will create only a minimal Jess engine, including a relatively small number of functions (the *intrinsic* functions). Many of the functions packaged with Jess are actually implemented as optional `jess.Userpackage` classes. You can choose to load some or all of these into your Jess applications. Omitting unneeded functions may be especially important in applets, where a small footprint is important.

Here is a snippet of code (from `jess/Main.java`) which will load all the standard optional functions, without causing an error if any of them are missing:

```
  // Load in optional packages, but don't fail if any are missing.
  Rete rete = new Rete(nd);

  String [] packages = { "jess.StringFunctions",
                         "jess.PredFunctions",
                         "jess.MultiFunctions",
                         "jess.MiscFunctions",
                         "jess.MathFunctions",
                         "jess.BagFunctions",
                         "jess.reflect.ReflectFunctions",
                         "jess.view.ViewFunctions" };

  for(int i=0; i< packages.length; i++)
    {
      try
```

```
        {
          rete.addUserpackage((Userpackage)
                              Class.forName(packages[i]).newInstance());
        }
      catch (Throwable t) { /* Optional package not present */ }
    }
```

To find out which functions are built into Jess and which are optional, see the Jess Function Guide, which lists the package in which each function appears.

5.1.3 Executing Individual Commands

For somewhat more control over Jess from your Java program, you can use the `Rete` class's `executeCommand(String cmd)` method. For example, after the above code, you could include the following:

```
  try
    {
      rete.executeCommand("(reset)");
      rete.executeCommand("(assert (foo bar foo))");
      rete.executeCommand("(run)");

      // Prints '42'
      System.out.println(rete.executeCommand("(+ 37 5)"));
    }
  catch (ReteException ex)
    {
      System.err.println("Foo bar error.");
    }
```

Commands executed via `executeCommand()` may refer to Jess variables; they will be interpreted in the *global context*. In general, only defglobals can be used in this way.

5.1.4 Other Methods in the `Rete` Class

`Rete` has a number of other public methods which I have not documented here; see the source for details. There are functions to assert and retract facts (which you are invited to use) and functions to find and remove various constructs (which you may also use.) There are also functions which allow you to add constructs (defrules, deftemplates, etc.) In general, it is best that you *not* use these in your programs, because they are highly subject to change in future Jess versions, as are the details of the classes like `jess.Defrule` which represent constructs. Two other functions you will certainly use are `addUserfunction()` and `addUserpackage()`. These are explained in Extending Jess With Commands Written in Java. In addition, the methods `run()`, `reset()`, and `halt()` are useful.

There is a set of diagnostic methods in Rete which return Enumerations of various data structures in the engine:

- public synchronized Enumeration listDeftemplates()
- public synchronized Enumeration listDefrules()
- public synchronized Enumeration listFacts()
- public synchronized Enumeration listActivations()
- public synchronized Enumeration listDefglobals()
- public synchronized Enumeration listUserfunctions()

Feel free to use these for debugging purposes, but don't get too chummy with them: they are subject to change without notice, especially the types of the objects they return. Note that `listUserfunctions()` returns an Enumeration of *all* the Jess functions in the engine, including deffunctions and intrinsics.

5.2 The `ReteDisplay` Interface.

There is a lot you can do with just what we've discussed so far. One thing that you might not like, though, is that by default, programs we write using the above techniques will send their output to your Java program's `System.out` and take input from `System.in`. If you're writing a graphical program, this is clearly not what you want. Jess provides an interface `jess.ReteDisplay` that lets you deal with this in a simple way. `ReteDisplay` also provides some hooks into the engine's internal workings. The `ReteDisplay` interface provides two types of functions:

1. Functions that return the intial input, output, and error streams that the engine should use.
2. Functions that are called by the engine whenever an event occurs (events here meaning that a construct is parsed, fact is asserted, rule is activated, and so forth).

5.2.1 Using ReteDisplay

Every `Rete` object must be constructed with an instance of `ReteDisplay.` Here is the definition of the `ReteDisplay` interface:

```
public interface ReteDisplay
  {
    // Rete sets its initial input/output using these
    // These must be implemented
    public java.io.PrintStream stdout();
    public java.io.InputStream stdin();
    public java.io.PrintStream stderr();

    // These notify the ReteDisplay object when things happen
    // Can do nothing if you want!
    public void assertFact(ValueVector fact);
    public void retractFact(ValueVector fact);
    public void addDeffacts(Deffacts df);
    public void addDeftemplate(Deftemplate dt);
    public void addDefrule(Defrule rule);
    public void activateRule(Defrule rule);
    public void deactivateRule(Defrule rule);
    public void fireRule(Defrule rule);

    // This gives the Rete object access to Applet resources
    // Should return null if not in an Applet
    public java.applet.Applet applet();
  }
```

Jess ships with two different classes that implement this interface. You can write your own or modify these for use in your own applications.

- The class `jess.NullDisplay`, which implements part of the Jess command-line interface, uses the notification functions as a means of notifying Java Observers of changes to the Rete network (`NullDisplay` extends `java.util.Observable`). This is an important part of the implementation of the `view` command, which can display the Rete network architecture interactively as rules are added, which can be a valuable debugging tool. `jess.NullDisplay` simply routes Java's standard output and standard input to the Rete engine. It is convenient to use for many applications because it has no GUI. We used it above in our simple examples, and the standard Jess command line driver `jess.Main` uses it as well.
- The class `jess.ConsoleDisplay` is more complex. It does everything `jess.NullDisplay` does, and more. It is a graphical display, routing Jess input and output into graphical text boxes. It uses the notification functions to support the `view` command just as NullDisplay does. The `jess.Console` and `jess.ConsoleApplet` classes are examples of programs that use `jess.ConsoleDisplay`. See Jess43/examples/console.html for an example Web page embedding `jess.ConsoleApplet`. These classes should give you a good idea of how to use `ReteDisplay` to embed Jess in a GUI application.

5.2.2 The `jess.TextAreaOutputStream` and `jess.TextInputStream` Classes

Jess ships with two utility classes that are very useful when building GUIs for Jess: the `jess.TextAreaOutputStream` and `jess.TextInputStream` classes. Both can serve as adapters between Jess and graphical input/output widgets. The TextAreaOutputStream class is, as the name implies, a Java OutputStream that sends any data written to it to a `java.awt.TextArea`. This lets you place Jess's output in a scrolling window on your GUI. The `jess.Console` and `jess.ConsoleApplet` jess GUIs use these classes. To use `TextAreaOutputStream`, simply write a class that implements `ReteDisplay`, returning a suitably wrapped `TextAreaOutputStream` from the `stdout()` and `stderr()` methods:

```
import jess.*;
  public class MyDisplay implements ReteDisplay
    {
       public TextArea ta = new TextArea(20, 80);
       TextAreaOutputTream taos = new TextAreaOutputStream(ta);
       PrintStream ps = new PrintStream(taos, true);

       public MyDisplay()
       {
         Frame f = new Frame("Jess Demo");
         f.add(ta);
         f.pack();
         f.show();
       }

       PrintStream stdout() { return ps; }
```

// ... (Plus all the other ReteDisplay methods) Now if you construct a `jess.Rete` object with an instance of this display, the Jess output will go into a scrolling window. Study `jess/ConsoleDisplay.java` and `jess/Console.java` to see a complete example of this.

`jess.TextInputStream` is similar, but it is an `InputStream` instead. It is actually quite similar to `java.io.StringBufferInputStream`, except that you can continually add new text to the end of the stream (using the `appendText()` method). It is intended that you create a `jess.TextInputStream`, return it from your `jess.ReteDisplay.stdin()` method, and then (in an AWT event handler, somewhere) append new input to the stream whenever it becomes available. See the same `jess/Console*` files for a complete usage example for this class as well.

5.2.3 Switching Streams in Mid-Stream

The Rete engine will call the `stdout()` and `stdin()` methods of your `ReteDisplay` class *at least once* when it is created. It is from these return values that the initial definition of the `t`, `WSTDOUT`, and `WSTDERR` I/O routers are made (see Manipulating Jess I/O Routers in Java). It will never call these methods again. This means that changing the return values of these functions over time may have no effect. To re-route input and output you must use the explicit router functions.

5.3 Manipulating Jess I/O Routers in Java

`ReteDisplay` lets you set up the minumal initial state for a `Rete` object, but Jess can read from more that just standard input and standard output. Jess I/O routers can be easily manipulated from Java. These six functions in the Rete class manipulate the router list:

- `public void addInputRouter(String s, InputStream is, boolean console)`
- `public void addOutputRouter(String s, OutputStream os)`
- `public void removeInputRouter(String s)`
- `public void removeOutputRouter(String s)`
- `public InputStream getInputRouter(String s)`
- `public OutputStream getOutputRouter(String s)`

When Jess starts up, there are one input router and three output routers defined: the `t` router, which reads and writes from the standard input and output; the `WSTDOUT` router, which Jess uses for all prompts, diagnostic outputs, and other displays; and the `WSTDERR` router, which Jess uses to print stack traces and error messages. You can reroute these inputs and outputs simply by changing the Input and Output streams they are attached to using the above functions. You can use any kind of streams you can dream up: network streams, file streams, etc.

The `boolean` argument `console` to the `addInputRouter` method specifies whether the stream should be treated like the standard input or like a file. The difference is that on console-like streams, a `read` call consumes an entire line of input, but only the first token is returned; while on file-like streams, only the characters that make up each token are consumed on any one call. That means, for instance, that a `read` followed by a `readline` will consume two lines of text from a console-like stream, but only one from a file-like stream, given that the first line is of non-zero length. This odd behaviour is actually just following the behaviour of CLIPS.

The `Rete` class has two more handy router-related methods: `outStream()` and `errStream()`, both of which return a `PrintStream` object. `outStream()` returns a stream that goes to the same place as the current setting of `WSTDOUT`; `errStream()` does the same for `WSTDERR`.

You can add your own routers which do I/O through any Java streams; they will immediately be usable from Jess. Look at the implementation of the `socket` Userfunction in `jess/MiscFunctions.java` for an idea of what's possible.

5.4 Calling Jess Functions and Getting Results Back

So far we have treated the `Rete` class more or less as a black box. We have only poked and prodded it from the outside without exchanging any real data, only strings of text. To achieve a tight integration between Jess and your application, you'll need to be able to go further. Most likely, you'll want to execute commands in the Jess language that accept arguments that cannot be easily or correctly represented in the String argument to `executeCommand()`. For example, a floating-point number or a Java object or a multifield value or a fact. It's actually quite easy to call any Jess function, pass in any data as arguments, and receive the result not as a string but as arbitrary data. To do this, you're going to have to learn about two more Jess classes: `jess.Value` and `jess.ValueVector`.

5.4.1 The class `jess.Value`

A `Value` is a self-describing data object. Every datum in Jess is contained in one. Once it is constructed, a `Value`'s type and contents cannot be changed. `Value`supports a `type()` function, which returns one of these type constants (defined in the class `jess.RU,` (RU = "Rete Utilities")):

```
  final public static int NONE              =     0; ; an empty value (not NIL)
  final public static int ATOM              =     1; ; a symbol
  final public static int STRING            =     2; ; a string
  final public static int INTEGER           =     4; ; an integer
  final public static int VARIABLE          =     8; ; a variable
  final public static int FACT_ID           =    16; ; a fact index
  final public static int FLOAT             =    32; ; a double float
  final public static int FUNCALL           =    64; ; a function call
  final public static int ORDERED_FACT      =   128; ; an ordered fact
  final public static int UNORDERED_FACT    =   256; ; a deftemplate fact
  final public static int LIST              =   512; ; a multifield
  final public static int DESCRIPTOR        =  1024; ; (internal use)
  final public static int EXTERNAL_ADDRESS =  2048; ; a Java object
  final public static int INTARRAY          =  4096; ; (internal use)
  final public static int MULTIVARIABLE     =  8192; ; a multivariable
  final public static int SLOT              = 16384; ; (internal use)
  final public static int MULTISLOT         = 32768; ; (internal use)
```

`Value` objects are constructed by specifying the data and the type. Each overloaded constructor assures that the given data and the given type are compatible. Note that for each constructor, more than one value of the `type` parameter is acceptable.

The available constructors are:

```
  public Value(Object o, int type) throws ReteException
  public Value(String s, int type) throws ReteException
  public Value(Value v)
  public Value(ValueVector f, int type) throws ReteException
  public Value(double d, int type) throws ReteException
  public Value(int value, int type) throws ReteException
  public Value(int[] a, int type) throws ReteException
```

`Value` supports a number of functions to get the actual data out of a `Value`object. These are

```
  public Object externalAddressValue() throws ReteException
  public String stringValue() throws ReteException
  public ValueVector factValue() throws ReteException
  public ValueVector funcallValue() throws ReteException
  public ValueVector listValue() throws ReteException
  public double floatValue() throws ReteException
  public double numericValue() throws ReteException
  public int descriptorValue() throws ReteException
  public int factIDValue() throws ReteException
  public int intValue() throws ReteException
  public int[] intArrayValue() throws ReteException
```

If you try to convert random values by creating a Value and retrieving it as some other type, you'll generally get a ReteException. However, some types can be freely interconverted: for example, integers and floats.

5.4.2 The class `jess.ValueVector`

Facts, function calls, lists, etc. are stored by Jess in objects of class `jess.ValueVector. ValueVector` is an extensible array of `Value` objects. You set an element of a `ValueVector` with `void set(Value, int)` and get an element with `Value get(int). set()` and `get()` will throw an exception if the index you're accessing is past the end of the current array. You can add a value to the end of a `ValueVector` with `void add(Value)` (which can extend the length of the internal data structures). `int size()` returns the actual number of `Value`s in the `ValueVector. void setLength(int)` lets you cheat by extending the length of a `ValueVector` to include null `Values.` (This is necessary sometimes to allow filling in many elements in random order.)

Facts (type `RU.ORDERED_FACT` or `RU.UNORDERED_FACT`) are stored as a ValueVector with the slots filled in a special way, as follows (the constants representing slot numbers *must* be used, as they may change):

SLOT NUMBER	**TYPE**	**DESCRIPTION**
RU.CLASS	RU.ATOM	The *head* or first field of the fact
RU.ID	RU.FACT_ID	The fact-id of this fact
RU.DESC	RU.DESCRIPTOR	One of `RU.ORDERED_FACT` or `RU.UNORDERED_FACT`
RU.FIRST_SLOT	(ANY)	Value of the first slot of this fact
RU.FIRST_SLOT + 1	(ANY)	second
...	...	...

Note that for ordered facts, the slots are stored in the order in which they appear, but in unordered (deftemplate) facts, they appear in the order given in the corresponding deftemplate.

Function calls (RU.FUNCALLs) are simpler; the first slot is the functor as an `RU.ATOM`, and all remaining slots are arguments. And multifields are even simpler: each element of the `ValueVector` is an element of the multifield.

5.4.3 Creating and Executing RU.FUNCALL Objects

Given the above, it is very easy to create and execute objects that represent Jess function calls in Java. This allows you to pass arbitrary arguments, and to receive the return value. The steps to follow are:

1. Create a `jess.Funcall` object (a subclass of `ValueVector`)
2. Append the desired arguments as `Value` objects
3. Call `Funcall.simpleExecute()` to execute the command.
4. The return value is a `Value` object containing the result.

For example, here we are calling the Jess `bind` function, to make the defglobal named `*x*` point to a Java object of the imaginary type `Foo.`

```
Rete engine = ...;
Foo foo = new Foo();

// Define the defglobal
engine.executeCommand("(defglobal ?*x* = 0)");

// Tell Jess not to reset defglobals when the reset command is issued
// Otherwise ?*x* will be reset to 0 and the Foo object will disappear
engine.executeCommand("(set-reset-globals FALSE)");

// Create the Funcall object
Funcall f = new Funcall("bind", engine);

// Now add the arguments, in left-to-right order. Each argument
// is a jess.Value object. Note the use of the
// RU.EXTERNAL_ADDRESS  type to pass in the Java object
f.add(new Value("*x*", RU.VARIABLE));
f.add(new Value(foo, RU.EXTERNAL_ADDRESS));

// Now execute the function (simpleExecute is a static function.)
f.simpleExecute(f, engine.globalContext());

// Now Jess code has access to the Java object!
engine.executeCommand("(printout t ?*x* crlf)");
```

There are a few restrictions on what you can do with this technique:

- You can't pass in nested function calls. Each `Funcall` must represent a single function to execute.
- Variable names can only be passed to the "bind" function; if you need to use variable names, use `executeCommand()` instead (it returns the result of executing the function, as a `jess.Value.`)

5.4.4 Using `store` and `fetch` to transfer Java objects

Jess 4.1 introduced a new set of functions to let you easily send Java objects between Jess and Java code. Basically, each `Rete` object now contains a special Hashtable that both Jess code and Java code can store objects into. If you `store` something from Java code, you can subsequently `fetch` it from Jess, and vice-versa, providing a very easy-to-use mechanism for communicating inputs and results between the two languages.

These methods are available in the class `Rete:`

```
    public Value store(String name, Value val);
    public Value store(String name, Object val);
    public Value fetch(String name);
```

while these methods are available in Jess:

```
    (store <name> <value>)
```

```
(fetch <name>)
```

Both `store` methods accept a "name" and a value (in Java, either in the form of a `jess.Value` object (see section 5.4.1 for details) or an ordinary Java object; in Jess, any value), returning any old value with that name, or null (or nil) if there is none. Both `fetch` methods accept a name, and return any value stored under that name, or null/nil if there is no such object. `store` and `fetch` are now the preferred way to send Java objects between Jess and Java. A complete example, with a main program in Java and a set of rules that return a result, is in the directory Jess43/jess/examples/xfer/ .

5.5 Using Jess in a Multithreaded Program

Jess is getting closer to being entirely thread-safe, as long as you use it in a reasonable fashion. One major difference between Jess and CLIPS is that you can call `run` from a rule RHS and have a new rule fired up in the middle of RHS execution! This would almost certainly cause problems, so you probably shouldn't do it.

Jess maintains separate mutexes for RHS execution and for LHS execution; in other words, only one assert or retract may be going on in a single engine at once, and only one rule RHS may be executing at once; however, these two activities may occur simultaneously.

If you use Java object matching on rule LHSs, be aware of interactions between multiple threads. It is certainly possible to come up with combinations of objects and rule engines that will deadlock; the easiest way to do this is to trigger a `PropertyChangeEvent` from a rule LHS, so be careful not to do that!

Less obvious but more pernicious is the fact that calling functions on rule LHSs which can directly or indirectly have the side-effect of asserting, retracting, or modifying a fact (this includes definstance and undefinstance, and modifying the properties of a definstanced Bean) can cause Jess to operate incorrectly. Again, be careful not to do this!

6 Extending Jess With Commands Written in Java

Jess's rule language can be extended with additional commands written in Java. For many real applications, extending Jess in this way may be necessary. The good news is that it's very easy, and you can add capabilities to Jess limited only by your imagination.

6.1 Extension Objects

The Java interface `jess.Userfunction` represents a single user-supplied function, while the interface `jess.Userpackage` represents a whole set of such functions. When you write a new function for Jess, you do it by writing a class that implements the `jess.Userfunction` interface (see below for details on how this is done.) Then a single instance of this class is created and installed into Jess. These objects can maintain state and can be retrieved by other code you write (see Obtaining References to Userfunction Objects). Therefore a Userfunction can cache results across invocations, maintain complex data structures, or keep references to external Java objects for callbacks.

6.2 Installing Extensions

Given that you have written or obtained some classes which implement these interfaces, you can load these extensions into Jess in two ways. First, you can load them in from Java code. Given that `rete` is the Rete object in your application, and `MyFunction` is the name of a Userfunction class you (or someone else!) wrote, you can add the new function to Jess by calling

```
rete.addUserfunction(new MyFunction());
```

or an entire package of such functions in a class `mypackage` using

```
rete.addUserpackage(new MyPackage());
```

You can also load extension functions and packages from the Jess language itself. The equivalents to the above are

```
(load-function "MyFunction")
```

and

```
(load-package "MyPackage")
```

Note that if the new classes or user packages come in a Java package, you'll need to specify the fully qualified name of the class:

```
(load-package "xyzzy.bassomatic.MyPackage")
```

In any case, the relevant classes need to be reachable on your Java `CLASSPATH`.

6.3 Writing Extensions

I've made it as easy as possible to add user-defined functions to Jess. There is no system type-checking on arguments, so you don't need to tell the system about your arguments, and values are self-describing, so you don't need to tell the system what type you return. You do, however, need to understand several Jess classes: `jess.Value`, `jess.ValueVector`, and `jess.Funcall`, as previously discussed in Embedding Jess in a Java Program.

To implement the `jess.Userfunction` interface, you need to implement only two methods: `name()` and `call().` Here's an example of a class called 'MyUpcase' that implements the Jess function `my-upcase`, which expects a String as an argument, and returns the string in uppercase.

```
  import jess.*;
  public class MyUpcase implements Userfunction
    {
      // The name method returns the name by which the function will appear in Jess
code.
      public String name() { return "my-upcase"; }

      public Value call(ValueVector vv, Context context) throws ReteException
        {
          return new Value(vv.get(1.)stringValue(.)toUpperCase(), RU.STRING);
        }
    }
```

The `call()` method does the business of your Userfunction. When `call()` is invoked, the first argument will be a `ValueVector` representation of the Jess code that evoked your function. For example, if the following Jess function call was made,

```
        (my-upcase "foo")
```

the first argument to `call()` would be a `ValueVector` of length two. The first element would be a `Value` containing the symbol (type RU.ATOM) `my-upcase`, and the second argument would be a `Value` containing the string (RU.STRING) `"foo"`.

Note that we use `vv.get(1).stringValue()` to get the first argument to `my-upcase` as a Java String. If the argument doesn't contain a string, or something convertible to a string, a `ReteException` may be thrown that describes the problem; hence you don't need to worry about incorrect argument types if you don't want to. `vv.get(0)` will always return the symbol `my-upcase`, the name of the function being called (the clever programmer will note that this would let you construct multiple objects of the same class, implementing different functions based on the name of the function passed in as a constructor argument); `vv.get(1)` is the first argument, and `vv.get(2)` would be the second, if this function accepted multiple arguments. If you want, you can check how many arguments your function was called with and throw a ReteException if it was the wrong number by using the `vv.size()` method. In any case, our simple implementation extracts a single argument and uses the Java `toUpperCase()` method to do its work. `call()` must wrap its return value in a `jess.Value` object, specifying the type (here it is `RU.STRING`).

Having written this class, you can then, in your mainline code, simply call `Rete.addUserfunction()` with an instance of your new class as an argument, and the function will be available from Jess code. Adding to our mainline code from the last section:

```
  // Add the 'my-upcase' command to Jess
  rete.addUserfunction(new MyUpcase());
  // Exceute some Jess code that calls this function
  rete.executeCommand("(printout t (my-upcase foo) crlf)");
```

will print `FOO`.

6.4 Writing Extension Packages

The `jess.Userpackage` interface is a handy way to group a collection of `Userfunctions` together, so that you don't need to install them one by one (all of the extensions shipped with Jess are included in Userpackage classes). A `Userpackage` class should supply the one method `add()`, which adds a collection of `Userfunctions` to a Rete engine using `addUserfunction()`. Nothing mysterious going on, but it's very convenient. As an example, the string utilities package `jess/StringFunctions.java` looks something like this:

```
  public class StringFunctions implements Userpackage
    {
      public void add(Rete engine)
        {
          engine.addUserfunction(new strcat());
          engine.addUserfunction(new upcase());
          engine.addUserfunction(new lowcase());
          engine.addUserfunction(new strcompare());
          engine.addUserfunction(new strlength());
          engine.addUserfunction(new substring());

        }
```

Now in your mainline, you can call

```
  engine.addUserpackage(new jess.StringFunctions());
```

and from your Jess code, you can call `str-cat`, `str-compare`, etc.

`Userpackage`s are a great place to assemble a collection of interrelated functions which potentially can share data or maintain references to other function objects. You can also use `Userpackage`s to make sure that your `Userfunctions` are constructed with the correct constructor arguments.

There are a lot of small Userfunction classes in the jess package which you can use as examples for writing your own Jess extensions. All of these small classes can add a lot of size overhead to Applets, which is why they are not all built-in to the engine. These days, with zips and JAR files, this isn't such a big deal. Still, you can leave them out if you want just by not adding the relevant Userpackage from your mainline program. Contrarily, if you don't add these packages to your programs, the corresponding Jess functions will not be available.

6.5 Obtaining References to Userfunction Objects

Occasionally it is useful to be able to obtain a reference to an installed `Userfunction` object. The method `Userfunction Rete.findUserfunction(String name)` lets you do this easily. It returns the `Userfunction` object registered under the given name, or null if there is none. This is most useful when you write Userfunctions which themselves maintain state of some kind, and you need access to that state.

---

7 Accessing Java Objects Directly from Jess

As of Jess 4.0, you can create Java objects, call their methods, and access their data directly from Jess *without writing any Java code!* The Jess functions that enable this are in the optional Userpackage `jess.reflect.ReflectFunctions` and they require Java 1.1 or later to compile. See the Jess Function Guide for a description of the `call`, `new`, `set`, `get`, `set-member` and `get-member` methods.

This new capability makes Jess more than just an expert system shell; it is now also a dynamic, extensible, portable

Java-based scripting environment.

7.1 Creating Java Objects

The Jess `new` function lets you create Java objects. The first argument is the fully-qualified name of the class as a symbol or String; any remaining arguments are passed to the Java object's constructor. For example:

```
(new java.lang.StringBuffer 100)
```

will create an instance of `StringBuffer,` passing the integer `100` as a constructor argument.

In the case of overloaded constructors, the constructor will be chosen from among all constuctors for the named class with the given number of arguments based on a *first-best fit* algorithm. Built-in Jess types are converted as necessary to match the available constructors. The Jess atoms `TRUE` and `FALSE` are automatically converted to Java booleans, while the atom `nil` is automatically converted to the Java null pointer. Jess multifields are automatically converted to one-dimensional Java arrays; there is no way to represent a multidimensional Java array in Jess (if this becomes necessary, you can always write a Userfunction to call the appropriate constructor). Floating point values are converted to the best-matching floating-point type. External address types are unwrapped and passed directly as Java arguments. If you have trouble calling the correct constructor, you can often disambiguate between multiple constructors by using Java *wrapper objects*. For example, if the imaginary `Foo` class had the two constructors

```
Foo(double d);
Foo(float f);
```

you could specifically call the one that takes a float argument instead of the double argument like this: way:

```
(new Foo (new java.lang.Float 123.456))
```

7.2 Calling Java Methods

The Jess `call` function lets you call Java methods. The first argument to `call` is either a variable holding the Java object on which to call the method, or the name of the class (for a static method.) The second argument is the name of the method. The remaining arguments are the arguments to the method. Jess will find an appropriate method using the same techniques described in Creating Java Objects.

An example: to use the `void java.lang.StringBuffer.append(String s)` method directly, you can write:

```
(defglobal ?*str-buf* = (new java.lang.StringBuffer 100))
(call ?*str-buf* append "Some String Data To Append")
```

Note that in many cases, explicit use of the `call` functor is optional; it can be omitted in function calls that are not nested inside of other function calls. For example, the above call to append could also be written as:

```
(?*str-buf* append "Some String Data To Append")
```

A static method example: you can invoke the Java garbage collector using the `java.lang.System.gc()` method like this:

```
(call java.lang.System gc)
```

7.3 Setting and Reading Java variables

You can set and read the values of public Java member variables using the `set-member` and `get-member` functions. Each of these functions accepts either a Java object or a class name (in the case of static members) as the first argument. The second argument is the name of the field. `set-member` requires a third argument, the new value for the field. There is no mechanism for setting or getting a specific member of an array field; you can only set or read complete arrays, which are represented in Jess as multifield values.

An example: if `?pt` holds a `java.awt.Point` object, you can reference its `x` coordinate field like this:

```
(printout t "The value of x is " (get-member ?pt x))
```

7.4 Setting and Reading Java Bean Properties

As mentioned previously, Java objects can be explicitly pattern-matched on the LHS of rules, but only to the extent that they are *Java Beans*. A Java Bean is really just a Java object that has a number of methods that obey a simple naming convention for *Java Bean properties*. A class has a Bean property if, for some string *X* and type *T* it has either or both of:

- A method named `get`*`X`* which returns *T* and accepts no arguments; or, if *T* is boolean, named `is`*`X`* which accepts no arguments;
- A method named `setX` which returns void and accepts a single argument of type *T*.

Note that the capitalization is also important: for example, for a method named isVisible, the property's name is *visible*, with a lower-case `V`. Only the capitalization of the first letter of the name is important. You can conveniently set and get these properties using the Jess `set` and `get` methods. Note that many of the trivial changes in the Java 1.1 were directed towards making most visible properties of objects into Bean properties.

An example: AWT components have many Bean properties. One is *visible*, the property of being visible on the screen. We can query this property in two ways: either by explicitly calling the `isVisible()` method, or by querying the Bean property.

```
(defglobal ?*frame* (new java.awt.Frame "Frame Demo"))

; Directly call 'isVisible', or...
(printout t (call ?*frame* isVisible) crlf)

; ... equivalently, query the Bean property
(printout t (get ?*frame visible) crlf)
```

7.5 Writing Java GUIs in Jess

One special case of manipulating Java objects directly from Jess is the building of Java GUIs from Jess code. I think this will be pretty common, so I've provided some utilities for extending what is possible. All of these utility classes are in the `jess.reflect` package.

It should now be obvious that you can easily construct GUI objects from Jess. For example, here is a `Button`:

```
(defglobal ?*b* = (new java.awt.Button "Hello"))
```

What should not be obvious is how, from Jess, you can arrange to have something happen whan the button is pressed. For this, I have provided a full set of `EventListener` classes:

- `jess.reflect.ActionListener`
- `jess.reflect.AdjustmentListener`
- `jess.reflect.ComponentListener`
- `jess.reflect.ContainerListener`
- `jess.reflect.FocusListener`
- `jess.reflect.ItemListener`
- `jess.reflect.KeyListener`
- `jess.reflect.MouseListener`
- `jess.reflect.MouseMotionListener`
- `jess.reflect.TextListener`
- `jess.reflect.WindowListener`

Each of these classes implements one of the `Listener` interfaces from the `java.awt.event` package in Java 1.1. Each implementation packages up any event notifications it receives and forwards them to a Jess `deffunction`, which is supplied as a constructor argument to the `Listener` object.

An example should clarify matters. Let's say that when the `Hello` button is pressed, you would like the string `Hello, World!` to be printed to standard output (how original!). What you need to do is:

1. Define a `deffunction` which prints the message. The `deffunction` will be called with one argument: the event object that would be passed to `actionPerformed()`. (If this is gibberish to you, pick up a book on Java AWT programming.)
2. Create a `jess.reflect.ActionListener` object, telling it about this `deffunction`, and also which Jess engine it belongs to.
3. Tell the `Button` about the `ActionListener` using the `addActionListener` method of `java.awt.Button`.

Here's a complete program in Jess:

```
(defglobal ?*f* = (new java.awt.Frame "Button Demo"))
(defglobal ?*b* = (new java.awt.Button "Hello"))

(deffunction say-hello (?evt)
  (printout t "Hello, World!" crlf))

(?*b* addActionListener
  (new jess.reflect.ActionListener say-hello (engine)))

(?*f* add ?*b*)
(?*f* pack)
(set ?*f* visible TRUE)
```

The Jess `engine` function returns the `jess.Rete` object in which it is executed, as an external address. You'll have to quit using ^C. To fix this, you can add a `WindowListener` which handles `WINDOW_CLOSING` events to the above program:

```
(deffunction frame-handler (?evt)
  (if (= (?evt getID) (get-member ?evt WINDOW_CLOSING)) then
      (call (get ?evt source) dispose)
      (exit)))

(?*f* addWindowListener
  (new jess.reflect.WindowListener frame-handler (engine)))
```

Now when you close the window Jess will exit. Notice how we can examine the `?evt` parameter for event information.

See the demo `examples/frame.clp` for a slightly more complex example of how you can build an entire Java graphical interface from within Jess.

7.6 Screen Painting and Graphics from Jess

As you may know, the most common method of drawing pictures in Java is to subclass `java.awt.Canvas`, overriding the `void paint(Graphics g)` method to call the methods of the `java.awt.Grpahics` argument to do the drawing. Well, Jess can't help you to subclass a Java class (at least not yet!), but it does provide an adaptor class, much like the event adaptors described above, that will help you draw pictures. The class is named `jess.reflect.Canvas`. When you construct an instance of this class from Jess, you pass in the name of a deffunction (which will do the painting) and a reference to the Rete engine. Whenever `paint()` is called to render the `jess.reflect.Canvas,` the `jess.reflect.Canvas` in turn will call the deffunction. The deffunction will be passed two arguments: the `jess.reflect.Canvas` instance itelf, and the `java.awt.Graphics` argument to `paint().` In this way, Jess code can draw pictures using Java calls. An example looks like this:

```
;; A painting deffunction. This function draws a red 'X' between the
;; four corners of the Canvas on a blue field.

(deffunction painter (?canvas ?graph)
  (bind ?x (get-member (call ?canvas getSize) width))
```

```
    (bind ?y (get-member (call ?canvas getSize) height))
    (?graph setColor (get-member java.awt.Color blue))
    (?graph fillRect 0 0 ?x ?y)
    (?graph setColor (get-member java.awt.Color red))
    (?graph drawLine 0 0 ?x ?y)
    (?graph drawLine ?x 0 0 ?y))

  ;; Create a canvas and install the paint routine.
  (bind ?c (new jess.reflect.Canvas painter (engine)))
```

A simple but complete program built on this example is in the file `examples/draw.clp` in the Jess distribution.

7.7 Pattern Matching Java Beans

Java Beans is a component architecture for Java, added in Java 1.1. One of the most exciting recent additions to Jess is the ability to match Java Beans on rule LHSs as if they were facts. In fact, such objects *become* facts. When a Java object is properly registered for pattern-matching with Jess, Jess creates and maintains a fact on the fact-list that always represents the state of that object. It is these *shadow facts* that are actually matched on rule LHSs. In the following sections, we will learn how to tell Jess to create these shadow facts.

The process is fairly simple, and involves two steps. First, you tell Jess to build an internal deftemplate representing a particular Java class. You use the defclass construct to do this. Then you can tell Jess to shadow a specific object of that class using the definstance command.

I will try to use a minimum of Java Beans jargon in what follows, but if you are unfamiliar with the Java Beans architecture, you might want to grab a Java Beans book from your favorite computer bookstore before continuing.

7.7.1 The `defclass` construct

`defclass` is used to tell Jess that you would like to be able to use a specific Java class like a deftemplate. The syntax of `defclass` is simple:

```
  (defclass pump jess.examples.pumps.Pump)
```

The class tag `pump` is just like a deftemplate name. Jess automatically builds a deftemplate from the named Java class, in this case `jess.examples.pumps.Pump`, an example class shipped with Jess. Jess turns Java Beans properties into deftemplate slots. (If you're not familiar with Java Beans, this basically means that Jess will find any methods in the class named `get<X>` or `is<X>` which return non-void and take no arguments. Jess uses the BeanInfo mechanism to learn about Bean properties.) As an example, the `jess.examples.pumps.Pump` class looks something like this (lots cut out!):

```
  public class Pump
  {
    public String getName() { ... }
    public int getFlow() { ... }
    public void setFlow(int f) { ... }
  }
```

Jess will generate this deftemplate:

```
  (deftemplate pump "$JAVA-OBJECT$ jess.examples.pumps.Pump"
    (slot class)
    (slot name)
    (slot flow)
    (slot OBJECT))
```

The extra slot `class` comes from the `getClass()` method inherited from `java.lang.Object`. The `OBJECT` slot is added to every defclass by Jess: it always holds a reference to the Java object the matched pattern refers to. The `name` and `flow` slots are generated by Jess dueto the presence of the getName and getFlow methods. Jess uses the `java.beans.Introspector` class to determine the properties of a Bean, which means it respects BeanInfo objects, if present.

Array properties give rise to multislots in Jess.

There's a simple Java Bean class to experiment with in `jess.examples.simple.`

7.7.2 The `definstance` construct

`definstance` tells Jess to create shadow facts for a particular Java object. Once a `definstance` construct has been issued, there will always be a fact on the fact-list representing that object (until the `clear` command is issued.) The `reset` command refreshes, but does not remove, these facts.

The syntax of `definstance` is simple, too:

```
  (definstance <defclass-name> <external-address>)
```

`definstance` simply pairs Java objects (the <external-address> argument) with `defclass` names.

An object to be treated in this way must satisfy two requirements:

1. It must be assignable to a reference of the type defined in a previous `defclass`. In other words, it must be of the same class, a subclass, or an implementor of a defclassed interface.
2. It must support `PropertyChangeListeners` (an event mechanism used by Java Beans). In other words, it must support the `addPropertyChangeListener(PropertyChangeListener pcl)` method. Note that none of the classes in the Java API support `PropertyChangeListeners`; however, many Java Beans do, and support is easily added to other classes using the `java.beans.PropertyChangeSupport` class.

In any case, you can install an object for pattern matching like this:

```
  (definstance pump (new jess.examples.pumps.Pump "MAIN" ?tank))
```

where the first argument to `definstance` is a `defclass` tag, and the second is an expression returning the Java object. `"MAIN"` and `?tank` are arguments to the `jess.examples.pumps.Pump` constructor.

Here we are creating the object using the `new` function (see Creating Java Objects). It instead could come from a variable, from the return value of a Userfunction, and so forth. If you want to pattern-match on an object created from Java code, you can use the store and fetch facility, like this:

```
import jess.*;
import jess.examples.pumps.*;
...
  Rete engine = ...;
  Tank t = ...;
  Pump p = new Pump("MAIN", t);

  // Register the object with the engine under the name "pump-1"
  engine.store("pump-1", new Value(p, RU.EXTERNAL_ADDRESS));

  // Tell the definstance command to fetch the object named "pump-1"
  engine.executeCommand("(definstance pump (fetch pump-1))");
```

or you can use the technique from section 5.4.3, Creating and Executing RU.FUNCALL Objects, to pass the object to Jess. You could use this technique to set a defglobal to point to the object, as in the example in that section, or you could use it to call the definstance function directly, like this:

```
import jess.*;
import jess.examples.pumps.*;
...
  Rete engine = ...;
  Tank t = ...;
  Pump p = new Pump("MAIN", t);

  // Create the Funcall object
  Funcall f = new Funcall("definstance", engine);

  // Now add the arguments
  f.add(new Value("pump", RU.STRING));
  f.add(new Value(p, RU.EXTERNAL_ADDRESS));

  // Execute the function
  f.simpleExecute(f, engine.globalContext());
```

Once declared in a `definstance` construct, a Java object will be represented at all times by a single fact in Jess. This fact will be modified each time the object sends a `PropertyChangeEvent`. You can write patterns on the LHS of a rule to match this automatically generated and maintained fact, and as a result, you can match the state of the Java object. For example, given the `pump` `defclass` and `definstance`, we can write the following rule:

```
  (defrule decrease-pump-flow-if-high-1
    "If a pump's flow rate is over 20, decrease it by 5 units."
    (pump (flow ?f&:(> ?f 20)) (OBJECT ?pump))
    =>
    (set ?pump flow (- ?f 5)))
```

Note that by binding the value of the `OBJECT` slot on the LHS, we can modify the matched object from the RHS of the rule. You can also use the `modify` function for this purpose:

```
  (defrule decrease-pump-flow-if-high-2
    "If a pump's flow rate is over 20, decrease it by 5 units."
    ?pump <- (pump (flow ?f&:(> ?f 20)))
    =>
    (modify ?pump (flow (- ?f 5))))
```

These two rules are equivalent, although the latter may be slightly more efficient.

The Pump example given here is taken from a full, working example that ships with Jess. To try it out, compile the classes in `jess/examples/pumps`:

```
  javac jess\examples\pumps\*.java (Unix)
```

or

```
 javac jess/examples/pumps/*.java (Win32)
```

and then run the example as a Jess script:

```
  java jess.Main examples\pumps\pumps.clp (Unix)
```

or

```
  java jess.Main examples/pumps/pumps.clp (Win32)
```

or the alternate version in which the objects are created in Java:

```
  java jess.examples.pumps.MainInJava
```

(Of course, you also need to have first compiled the optional reflection commands in `jess\reflect.`) Read the Java source files and the `pumps.clp` file to see what's going on. It's a real-time control problem and Jess does a passable job of keeping two leaky tanks from overflowing or running dry. The Pumps and Tanks are Java objects that run in their own threads, while Jess reacts to their `PropertyChangeEvent`s, triggering rules which in turn adjust the pumps.

Note: You must be careful not to trigger any `PropertyChangeEvents` via calculations you perform on the LHS of any rule. A rule of thumb is not to set any Java variables or call any methods that might possibly change an objects state. Property accessor methods are fine. Violation of this warning can cause thread deadlock in the engine.

7.8 Efficiency

This reflection capability is extremely powerful and useful. However, such function calls will not be as efficient as calls made to the same Java function through a compiled Userfunction interface. For example, the two Jess function calls:

```
  (pi)
```

and

```
  (get-member java.lang.Math PI)
```

both return the value of the static final member `PI` of the class `java.lang.Math`, but the first call is much more efficient because `pi` is a Userfunction. If you are going to call a Java function just once, go ahead and use `call`, but if you're going to use it in a loop, consider wrapping it in a Userfunction if performance might be an issue.

8 How Jess Works

Note: The information in this Section is provided for the curious reader. An understanding of the Rete algorithm may be helpful in planning expert systems; an understanding of Jess's implementation probably will not. Feel free to skip this section and come back to it some other time. You should not take advantage of many of the Java classes mentioned in this section. They are internal implementation details and any Java code you write which uses them may well break each time a new version of Jess is released.

Jess is a rule-based expert system shell. In the simplest terms, this means that Jess's purpose it to continuously apply a set of if-then statements (*rules*) to a set of data (*fact list*). You define the rules that make up your own particular expert system. Jess rules look something like this:

```
  (defrule library-rule-1
    (book (name ?X) (status late) (borrower ?Y))
    (borrower (name ?Y) (address ?Z))
   =>
    (send-late-notice ?X ?Y ?Z))
```

Note that this syntax is identical to the syntax used by CLIPS. This rule might be translated into psueudo-English as follows:

```
   Library rule #1:
   If
     a late book exists, with name X, borrowed by someone named Y
   and
     that borrower's address is known to be Z
   then
     send a late notice to Y at Z about the book X.
```

The book and borrower entities would be found on the fact list. The fact list is therefore a kind of database of bits of factual knowledge about the world. The attributes (called *slots*) that things like books and borrowers are allowed to have are defined in statements called *deftemplates*. Actions like `send-late-notice` can be defined in user-written functions in the Jess language (`deffunctions`) or in Java (Userfunctions). For more information about the CLIPS rule syntax (and to work with Jess, you will certainly need to learn more!) refer to the previous section and to the CLIPS documentation mentioned earlier.

The typical expert system has a fixed set of rules while the fact list changes continuously. However, it is an empirical fact that, in most expert systems, much of the fact list is also fairly fixed from one rule operation to the next. Athough new facts arrive and old ones are removed at all times, the percentage of facts that change per unit time is generally fairly small. For this reason, the obvious implementation for the expert system shell is very inefficient. This obvious implementation would be to keep a list of the rules and continuously cycle through the list, checking each one's left-hand-side (LHS) against the fact list and executing the right-hand-side (RHS) of any rules that apply. This is inefficient because most of the tests made on each cycle will have the same results as on the previous iteration. However, since the fact list is stable, most of the tests will be repeated. You might call this the *rules finding facts* approach and its computational complexity is of the order of

O(RF^P), where R is the number of rules, P is the average number of patterns per rule LHS, and F is the number of facts on the fact list. This escalates dramatically as the number of patterns per rule increases.

Jess instead uses a very efficient method known as the Rete (Latin for *net*) algorithm. The classic paper on the Rete algorithm *("Rete: A Fast Algorithm for the Many Pattern/ Many Object Pattern Match Problem", Charles L. Forgy, Artificial Intelligence 19 (1982), 17-37)* became the basis for a whole generation of fast expert system shells: OPS5, its descendant ART, and CLIPS. In the Rete algorithm, the inefficiency described above is alleviated (conceptually) by remembering past test results across iterations of the rule loop. Only new facts are tested against any rule LHSs. Additionally, as will be described below, new facts are tested against only the rule LHSs to which they are most likely to be relevant. As a result, the computational complexity per iteration drops to something more like O(RFP), or linear in the size of the fact base. Our discussion of the Rete algorithm is necessarily brief. The interested reader is referred to the Forgy paper or to *Giarrantano and Riley, "Expert Systems: Principles and Programming", Second Edition, PWS Publishing (Boston, 1993)* for a more detailed treatment.

The Rete algorithm is implemented by building a network of nodes, each of which represents one or more tests found on a rule LHS. Facts that are being added to or removed from the fact list are processed by this network of nodes. At the bottom of the network are nodes representing individual rules. When a set of facts filters all the way down to the bottom of the network, it has passed all the tests on the LHS of a particular rule and this set becomes an *activation*. The associated rule may have its RHS executed (*fired*) if the activation is not invalidated first by the removal of one or more facts from its activation set.

Within the network itself there are broadly two kinds of nodes: one-input and two-input nodes. One-input nodes perform tests on individual facts, while two-input nodes perform tests across facts and perform the grouping function. Subtypes of these two classes of node are also used and there are also auxilliary types such as the terminal nodes mentioned above.

An example is often useful at this point. The following rules:

```
 (defrule example-2        (defrule example-3
    (x)                       (x)
    (y)                       (y)
    (z)                       => )
    => )
```

might be compiled into the following network:

```
        +----+  +----+  +----+     +----+   +----+  (one-input nodes)
        | x? |  | y? |  | z? |     | x? |   | y? |
        +----+  +----+  +----+     +----+   +----+
          \       /       |          \        /
        +------------+    |        +------------+
        |     +      |    |        |     +      |
        +------------+    |        +------------+
                  \       |              |          (two-input nodes)
                +------------+           |
                |     +      |           |
                +------------+           |
                      |                  |
            +----------------+     +----------------+
            | fire example-2 |     | fire example-3 |   (terminals)
            +----------------+     +----------------+
```

The nodes marked `x?`, etc., test if a fact contains the given data, while the nodes marked `+` remember all facts and fire whenever they've received data from both their left and right inputs. To run the network, Jess presents new facts to each node at the top of the network as they added to the fact list. Each node takes input from the top and sends its output downwards. A single input node generally receives a fact from above, applies a test to it, and, if the test passes, sends the fact downward to the next node. If the test fails, the one-input nodes simply do nothing. The two-input nodes have to integrate facts from their left and right inputs, and in support of this, their behavior must be more complex. First, note that any facts that reach the top of a two-input node could potentially contribute to an activation: they pass all tests that can be applied to single facts. The two input nodes therefore must remember all facts that are presented to them, and attempt to group facts arriving on their left inputs with facts arriving on their right inputs to make up complete activation sets. A two-input node therefore has a *left memory* and a *right memory*. It is here in these memories that the inefficiency described above is avoided. A convenient distinction is to divide the network into two logical components: the single-input nodes comprise the *pattern network*, while the two-input nodes make up the *join network*.

There are two simple optimizations that can make Rete even better, The first is to share nodes in the pattern network. In the network above, there are five nodes across the top, although only three are distinct. We can modify the network to share these nodes across the two rules (the arrows coming out of the top of the `x?` and `y?` nodes are outputs):

```
                    +--------------------------+
                    ^       +-------------+    |
                    |       ^             |    |
                  +----+  +----+  +----+  |    |
                  | x? |  | y? |  | z? |  |    |
                  +----+  +----+  +----+  |    |
                  /      /        /       |    |
        +------------+  /    +---/   +------------+
        |     +      |-+    /        |     +      |
        +------------+     /         +------------+
                  \       /              |
                +------------+           |
                |     +      |           |
                +------------+           |
                      |                  |
            +----------------+     +----------------+
            | fire example-2 |     | fire example-3 |
            +----------------+     +----------------+
```

But that's not all the redundancy in the original network. Now we see that there is one join node that is performing exactly the same function (integrating x,y pairs) in both rules, and we can share that also:

```
                  +----+  +----+  +----+
                  | x? |  | y? |  | z? |
                  +----+  +----+  +----+
                  /      /        /
        +------------+  /    +---/
        |     +      |-+    /
        +------------+     /
         |         \       /
         |       +------------+
         |       |     +      |
         |       +------------+
         |             |
         |   +----------------+
         |   | fire example-2 |
         |   +----------------+
   +----------------+
   | fire example-3 |
   +----------------+
```

The pattern and join networks are collectively only half the size they were originally. This kind of sharing comes up very frequently in real systems and is a significant performance booster!

You can see the amount of sharing in a Jess network by using the `watch compilations` command. When a rule is compiled and this command has been previously executed, Jess prints a string of characters something like this, which is the actual output from compiling rule example-2, above:

```
example-2: +1+1+1+1+1+1+2+2+t
```

Each time `+1` appears in this string, a new one-input node is created. `+2` indicates a new two-input node. Now watch what happens when we compile example-3:

```
example-3: =1=1=1=1=2+t
```

Here we see that `=1` is printed whenever a pre-existing one-input node is shared; `=2` is printed when a two-input node is shared. `+t` represents the terminal nodes being created. (Note that the number of single-input nodes is larger than expected. Jess creates separate nodes that test for the head of each pattern and its length, rather than doing both of these tests in one node, as we implicitly do in our graphical example.) No new nodes are created for rule example-3. Jess shares existing nodes very efficiently in this case.

Jess's Rete implementation is very literal. Different types of network nodes are represented by various subclasses of the Java class `jess.Node`: `Node1`, `Node2`, `NodeNot2`, `NodeTest`, and `NodeTerm`. The `Node1` class is further specialized because it contains a *command* member which causes it to act differently depending on the tests or functions it needs to perform. For example, there are specializations of `Node1` which test the first field (called the *head*) of a fact, test the number of fields of a fact, test single slots within a fact, and compare two slots within a fact. There are further variations which participate in the handling of multifields and multislots. The Jess language code is parsed by the class `jess.Jesp`, while the actual network is assembled by code in the class `jess.ReteCompiler`. The execution of the network is handled by the class `Rete`. The `jess.Main` class itself is really just a small demonstration driver for the jess package, in which all of the interesting work is done.

The `view` command is a graphical viewer for the Rete network itself; I have used this as a debugging tool for Jess, but it may have educational value for others, and it may help you to design more efficient systems of rules in Jess. Issuing the `view` command after entering the rules example-2 and example-3 produces a very good facsimile of the drawing (although it correctly shows the larger number of one-input nodes). The various nodes are color-coded according to their roles in the network; `Node1` nodes are red; `Node2` nodes are green; `NodeNot2` nodes are yellow; and `NodeTerm` nodes are blue. Passing the mouse over a node displays information about the node and the tests it contains; double-clicking on a node brings up a dialog box containing the same information (for join nodes, the memory contents are also displayed, while for `NodeTerm` nodes, a pretty-print representation of the the rule is shown). See the description of the `view` function for important information before using it.

8 The Future of Jess

Jess will continue to be maintained and improved for the foreseeable future. I have a list of features I plan to implement, but it's hard to associate timescales with any of them. They are listed in no particular order.

- Serialization; first, something like CLIPS `bload` and `bsave`; ultimately, the ability to save the compiled form of individual rules.
- More conflict resolution strategies, including user-definable ones.
- Backward chaining: a `goal` conditional element like ART.
- The `logical` conditional element.
- A much more extensive Java API for embedding Jess in other applications
- Optional compilation of Jess rules to pure Java code (potential for large speed improvements)

9 Version History

Version 4.3 (December 3rd, 1998):

Fixed redundant default-value processing, which was leading to odd problems with definstances with null slot values (thanks to S.S. Ozsariyildiz). Removed intern()s from Tokenizer (faster compilation). Fixed NIL/nil ambiguity in ReflectFunctions (thanks Andreas Rasmussen.) New web site; no longer distribute "index.html".

Version 4.2 (November 12th, 1998):

Fixed 'Corrupted negcnt' bug (thanks to Todd Bowers). (if ... then) function now throws an exception if atom 'then' is missing. Version string in 4.1 final was inadvertently left at 4.1b6. Added section to README explaining rule LHS semantics a bit better. Rete.findFactByID() is now public. Fix for very tricky 'phantom fact' problem reported by Steve Bucuvalas. Java method calls on Jess Strings now work for all Strings, not just alphanumeric ones. "animals" example modified to work with transitional gensym implementation.

Version 4.1 (September 15th, 1998):

Some minor bug fixes; code to allow you to leave off the '$' on a multivar after its first use, as in CLIPS.

Version 4.1b6:

Allow named variables in (not) CEs as long as they're not used in subsequent CEs. Fix a bug that was causing (return) to not work if inside a (foreach) inside a deffunction. Recursive deffunctions now work again. Jess works around a bug in some versions of Java that was preventing the atom '-' from parsing. Rete.listDefglobals() no longer lists duplicates of redefined defglobals (Karl Mueller found this one.) ReteDisplay.fireRule() is now called as appropriate. Accessing pattern-binding variables on rule LHSs works again (Karl again.) (reset) wasn't clearing all activations (thanks Al Davis); fixed. Funcall.toString() puts parens around the ValueVector version.

Version 4.1b5:

Just remove some debug code and extra files inadvertently shipped with 4.1b4.

Version 4.1b4:

addUserfunction, addDeffunction, etc collapsed into one addUserfunction routine in Rete class; same with findUserfunction. RU.getAtom() and RU.putAtom are gone! Userfunction.name() now returns String. ControlStructure interface used to clean up handling of such things. ReteCompiler uses Hashtables of Bindings instead of int[][] for vartables. Added default-dynamic deftemplate slot qualifier. Added set-/get-reset-globals, and changed the default defglobal reset behaviour. Added dynamic rule salience. Removed arbitrary limit of 32 slots for ordered facts and 32 tests per slot for all facts. "unique" CE (15-30% speed increase for many problems!) Various documentation improvements (many thanks to Win Carus.) Better error reporting (addContext() call in ReteException.) Malformed calls to 'eval' or 'build' or 'executeCommand' no longer go into an infinite loop on EOF. Added "store" and "fetch". Added "external-addressp". Rearranged Test1, Test2 classes into an inheritance hierarchy with a virtual doTest method, allowing for alternate implementations (undocumented java-test functionality included). Value class will do more type conversions automatically. Final multifield argument of a deffunction now acts as a wildcard, as in CLIPS (thanks David Young.)

Version 4.1b3

Problem with calling public methods of package-private classes from Jess fixed thanks to Lars Rasmusson's explanation. `OutOfMemoryError` while parsing file containing unbalanced open parens fixed. Line breaks in double-quoted strings no longer need to be (but can be) escaped. Two fixes thanks to Andreas Rasmusson: `gensym*` returns a symbol as documented, not a string; and a `propertyChangeEvent` for a bogus property no longer causes Jess to retract a `definstance` without updating it. Many of the synchronized methods in the `Rete` class no longer are synchronized; instead they use either synchronized blocks keyed to affected members or simply depend on the internal synchronization of Hashtables. `read` and `readline` explicitly act differently for console-like and file-like streams. `ConsoleDisplay` gets a *Clear Window* button.

Version 4.1b2

Bug in character-escape lexing fixed thanks to Josiah Poon. Parser-related bug in `explode$` fixed thanks to Andrew X. Comas. `eval`, `build`, `executeCommand()` again properly return the result of last expression, not EOF. `min`, `max` take arbitrary # of arguments. `implode$` now works; it apparently never really did. `printout` puts parens around multifields again. `str-compare` documentation corrected. `undefinstance` now removes the fact representing an object as well as deactivating matching. Wrote large regression test suite (not included in distribution). Bug in multiple simultaneous `Rete.run()` calls in separate threads fixed thanks to Andreas Rasmusson. Selectable conflict resolution strategies (only depth and breadth supported now) and user-definable strategies. The `try` command is added.

Version 4.1b1

Much better lexer (no more `StreamTokenizer`). Input buffering problems with JDK 1.1.2-1.1.4 fixed. Bug in (`test`) CE fixed. Can call `run` on rule RHS. Bug in incremental update fixed. Separate command-line, applet, and GUI console driver classes (`Quiz*` classes broken up, renamed to `Console*`). `read` and `readline` should work exactly as in CLIPS. Manual describes more about how to write Java `main()`. Bug in `definstance` that was preventing use of subclasses of a defclassed class is fixed.

Version 4.0

BeanInfo support. `quiz.html` embeds only one `QuizDisplay` applet. `Pumps` demo works again (sorry). Conflict resolution strategy now should be exactly the same as CLIPS's default.

Version 4.0b4

Extensive manual rewrite, adding lots of Java/Jess interoperation info. Allow standard CLIPS deffunction docstrings. Thanks to Jack Fitch, Dave Carlson and Alex Jacobson, property names for reflected Java Beans now use standard capitalization transform. Better error reporting, especially during parsing and from the command line. `set` and `get` renamed to `set-member` and `get-member`. `set` and `get` are now functions that read and write Bean properties. `ppdefrule` properly handles quoted strings in function calls. `executeCommand` and friends reuse a single parser. Thanks to Karl Mueller for `Rete.retractString`. Taught `batch` to read applet-based data files. `eval` now handles non-sexps. Better mechanism for synchronizing streams. `QuizDisplay` is an applet as well as an application. `run` accepts an argument, the maximum number of rules to fire. Fixed bug in `modify` when new slot value was a zero-length multifield. Fixed `ReteCompiler` bug where `MTELN` nodes were not consistently generated for zero-length multifield matches. Thanks to Sidney Bailin, fixed problem with accessing defglobals and variables bound to pattern indexes on rule LHSs. Added `get-var` function. Added `undefinstance`. `modify` and `retract` now handle definstance facts specially. Fixed some `doPPPattern` bugs (Dave Carlson again!).

Version 4.0b3

Added `jess.reflect` package containing `new`, `call`, `set`, and `get`. Added `JessListener` and its subclasses. added `engine`. Changed printing of external-addresses to include Java class name. Changed parser to accept variable names as Funcall heads (`call` is substitued, resulting in a runtime error if `call` is not installed). `and` and `or` functions now accept any values as arguments, not only funcalls. Added `foreach` control structure. Command prompt doesn't print `NIL` return values. Fixed another `not` bug (thanks to Sidney Bailin). Added matching of Java objects on rule LHSs: `definstance`, `defclass`. `TokenTree` now uses `sortcode % 101` as hash key, not the `sortcode` itself. All global classes moved into `jess` package. `Jess` class renamed `Main`.

Version 4.0b2

Cleaned up router/parser interactions. Jess will now read only one construct on a line of input (just like CLIPS). All Jess output now goes through `WSTDOUT` router, not through `ReteDisplay.stdout()`. Fixed bug whereby second and later references to subfields of multifields on the LHS of a rule would resolve to the whole multifield. `modify` can now properly handle multislots. `format` handles trailing spaces. Finally, parsing of integers: `2` is an `RU.INTEGER`, while `2.0` is an `RU.FLOAT`. Added `eval` and `list-function$`.

Version 4.0b1

Code reformat. Major performance enhancements (`Value` and `Funcall` recycling; Fastfunction interface; Rete memories are now btrees; `RU.CLEAR` tokens). `test` CE. Return-value constraints. `ppdefrule` thanks to Rajaram Ganeshan. Blank variables in `not` CEs. `system` blocks by default. readline fixed. `build` supported. logic for predicate functions in Rete network now precisely the same as for CLIPS. `QuizDisplay` demo. `while` and `if` accept boolean variables. Implied returns from `if` and `while` functions. Added `explode$`. Added I/O routers: `open`, `close`. Added `format`. Added `bag`.

Version 3.2

`system` and `integer` Userfunction classes renamed (Win95 filename capitalization problem!). Broken `delete$`, `insert$`, `replace$` fixed. `view` command added. Big if/then in Funcall class finally removed in favor of separate implementation classes for intrinsics, leading to a modest speed increase. Documentation vastly expanded! Added catch for `ArrayOutOfBoundsException` in command-line interface; no more crash on wrong number of arguments. Broken `evenp`, `oddp` fixed. `str-cat`, `sym-cat` made more general. Broken `sub-string` fixed. Big switch in `Node1` class replaced by separate classes, leading to a very modest speed increase.

Version 3.1

Added the `assert-string` and `batch` commands. Two bug fixes in multislot code (thanks to Nancy Flaherty). Added `undefrule` and the ability to redefine rules. Added the `system` function, although it doesn't work very well under Java. Public function `engine()` in `jess.Context` class allows you to do fancier things in Userfunctions. Added the non-standard `load-package` and `load-function` functions. Many new contributed functions packaged with Jess for doing math, handling multifields, and other neat stuff thanks to Win Carus. Added `time` (1 second resolution).

Version 3.0

A few code changes to accomodate Microsoft's Java compiler; Jess now compiles unchanged with JVC thanks to Mike Finnegan. Added `member$` multifield function. Added `clear` intrinsic thanks to Karl Mueller. Introduced a new way of handling `not` patterns which I think finally guarantees there are no more not-related bugs remaining! `load-facts`, which has been non-functional throughout the beta period, is working again. Documentation now explains unzipping and compiling a little better. Modified the way fact-id's are handled so that you can write `(retract 3)` to retract fact #3.

Version 3.0b2

Lots of bug reports and improvement suggestions from the field - thanks folks! All the reported bugs in the multifield implementation, and some residual odd behavior in the `not` CE, have been fixed. The `exit` command has been added. A command prompt has been added. The # character can now be used in symbols. The access levels on some methods in the `Rete` class have been opened up; `Rete` is no longer final. `nth$` is now 1-based, as it is in CLIPS. The `if` and `while` constructs now fire on `not FALSE` instead of `TRUE`. The `str-index` function has been fixed and added. Probably a few more things I'm forgetting here. Thanks for the input. Particular thanks to Nancy Flaherty, Joszef Toth, Karl Mueller, Duane Steward, and Michelle Dunn for reporting bugs fixed in this version; sorry if I left anyone out.

Version 3.0b1

First public release of Jess 3.0.

Version 3.0a3

UserPackage interface. Lots of new example UserFunctions for multifields, string, and predicates.

Version 3.0a2

Multislots! Also important bug fix: under certain circumstances, the Rete network compilation could fail 1) if `(not())` CEs occurred on the LHS of a rule, 2) new variables were introduced in that rule's patterns listed after the `(not())` CEs, and 3) these latter variables were tested (i.e., in a predicate constraint) on the LHS of the rule.

Version 3.0a1

Incremental reset. Watch activations. `gc()` in `LostDisplay`, `NullDisplay`. Multifields! All the Rete engine classes are now in a package named `jess`. Many classes and methods that should not be manipulated by clients are now package-private.

Version 2.2.1

Ken Bertapelle found another bug, which has been squashed, in the pattern network.

Version 2.2

Jess 2.2 adds a few new function calls (`load-facts`, `save-facts`) and fixes a serious bug (thanks to Ken Bertapelle for pointing it out!) which caused Jess to crash when predicate constraints were used in a certain way. Another bug fix corrected the fact that `retract` only retracted the first of a list of facts. Jess used to give a truly inscrutable error message if a variable was first used in a not CE (a syntax error); the current error message is much easier to understand. I also clarified a few points in the documentation.

Version 2.1

Jess 2.1 is *much* faster than version 2.0. The Monkey example runs in about half the time as under Jess 2.0, and for some inputs, the speed has increased by an order of magnitude! This is probably the last big speed increase I'll get. For Java/Rete weenies, this speed increase came from banishing the use of java.lang.Vector in Tokens and in two-input node memories. Jess is now within a believable interpreted Java/C++ speed ratio range of about 30:1. Jess 2.1 now includes rule salience. It also implements a few additional intrinsic functions: `gensym*`, `mod`, `readline`. Jess 2.1 fixes a bug in the way predicate constraints were parsed under some conditions by Jess 2.0. The parser now reports line numbers when it encounters an error.

Version 2.0

Jess 2.0 is intrinsically about 30% faster than version 1.0. The internal data structures changed quite a bit. The Rete network now shares nodes in the Join network instead of just in the pattern network. The current data structures should allow for continued improvement.

If anyone writes an emulation of a CLIPS function that Jess omits, *please* send it to me and I'll include it in the next release (with credit to you, of course).

At the time of this writing, Jess has many thousands of registered users. I have been very pleased by this response and have enjoyed working with many of Jess's more ambitious users. If you use Jess, and if you have comments, questions, or concerns, please don't hesitate to ask.

Finally, thanks to Gary Riley and the gang at NASA for writing the marvelous CLIPS in the first place!

```
Ernest Friedman-Hill                          Phone: (925) 294-2154
Distributed Computing                         FAX:   (925) 294-2234
Sandia National Labs                          ejfried@ca.sandia.gov
Org. 8920, MS 9214                    http://herzberg.ca.sandia.gov
PO Box 969
Livermore, CA 94550
```